Guilherme Rosas Borges

Bachelor of Science

Practical Isolated Searchable Encryption in a Trusted Computing Environment

Dissertation submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Engineering

Adviser: Bernardo Luís da Silva Ferreira, Researcher,
Faculdade de Ciências e Tecnologia
da Universidade NOVA de Lisboa

Co-adviser: Henrique João Lopes Domingos, Assistant Professor,
Faculdade de Ciências e Tecnologia
da Universidade NOVA de Lisboa

Examination Committee

Chairperson: Pedro Abílio Duarte de Medeiros
Raporteur: Miguel Nuno Dias Alves Pupo Correia

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

December, 2018

Practical Isolated Searchable Encryption in a Trusted Computing Environment

This document was created using the (pdf)LaTeX processor, based in the "novathesis" template[1], developed at the Dep. Informática of FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

To Sirius and Ivan.

Acknowledgements

First and foremost, I would like to sincerely thank my advisers, Bernardo and Henrique, without whom this thesis would not be possible. Their experience, availability, and knowledge, for which I am very much grateful, were fundamental in the production of this work, and I hope to make good use of their advice in my future.

I am also deeply grateful to João Leitão, whose advice and friendship were also essential to me, both as a person and as a researcher. I am indebted to all my colleagues and friends at the NOVA LINCS Computer Systems group, which provided not only helpful advice and lots of patience while reviewing my work, but also countless moments of fun. They include, but are not limited to, Gonçalo Tomás, Pedro Ákos Costa, and Pedro Fouto, with whom I hope to continue this adventure. I would also like to acknowledge my colleagues and co-authors from HASLab, Bernardo Portela and Tiago Oliveira, whose advice and tips while collaborating in our papers and projects proved insightful and useful. Moreover, I leave a note of acknowledgement to all professors and researchers I work with within NOVA LINCS, and with whom I hope to continue having positive interactions: Albert van der Linde, Carla Ferreira, Carmen Morgado, Hervé Paulino, João Costa Seco, João Lourenço, João Silva, José Legatheaux Martins, Ludwig Krippahl, Luís Caires, Cecília Gomes, Nuno Preguiça, Paulo Lopes, Pedro Medeiros and Vítor Duarte.

Finally, I sincerely thank everything that my parents Helena and Joaquim, my grandparents Abel, Adelaide, Joaquim, and Rosa, and Sirius and Ivan have done for me; for they were the ones who really made this work possible.

To conclude, I am also thankful for the financial support of the LightKone project (H2020 grant agreement ID 732505), and from FCT/MCTES, through the strategic project NOVA LINCS (UID/CEC/04516/2013) and project HADES (PTDC/CCI-INF/31698/2017).

Imagination will often carry us to worlds that never were. But without it we go nowhere.
Carl Sagan

Abstract

Cloud computing has become a standard computational paradigm due its numerous advantages, including high availability, elasticity, and ubiquity. Both individual users and companies are adopting more of its services, but not without loss of privacy and control. Outsourcing data and computations to a remote server implies trusting its owners, a problem many end-users are aware. Recent news have proven data stored on Cloud servers is susceptible to leaks from the provider, third-party attackers, or even from government surveillance programs, exposing users' private data.

Different approaches to tackle these problems have surfaced throughout the years. Naïve solutions involve storing data encrypted on the server, decrypting it only on the client-side. Yet, this imposes a high overhead on the client, rendering such schemes impractical. Searchable Symmetric Encryption (SSE) has emerged as a novel research topic in recent years, allowing efficient querying and updating over encrypted datastores in Cloud servers, while retaining privacy guarantees. Still, despite relevant recent advances, existing SSE schemes still make a critical trade-off between efficiency, security, and query expressiveness, thus limiting their adoption as a viable technology, particularly in large-scale scenarios.

New technologies providing Isolated Execution Environments (IEEs) may help improve SSE literature. These technologies allow applications to be run remotely with privacy guarantees, in isolation from other, possibly privileged, processes inside the CPU, such as the operating system kernel. Prominent example technologies are Intel SGX and ARM TrustZone, which are being made available in today's commodity CPUs.

In this thesis we study these new trusted hardware technologies in depth, while exploring their application to the problem of searching over encrypted data, primarily focusing in SGX. In more detail, we study the application of IEEs in SSE schemes, improving their efficiency, security, and query expressiveness.

We design, implement, and evaluate three new SSE schemes for different query types, namely Boolean queries over text, similarity queries over image datastores, and multimodal queries over text and images. These schemes can support queries combining different media formats simultaneously, envisaging applications such as privacy-enhanced

medical diagnosis and management of electronic-healthcare records, or confidential photograph catalogues, running without the danger of privacy breaks in Cloud-based provisioned services.

Keywords: Searchable Symmetric Encryption; Trusted Hardware; Cloud Computing; Privacy; Secure Boolean Querying; Secure Content-Based Image Retrieval

Resumo

O conceito de Computação na Nuvem tornou-se num paradigma de computação padrão, dadas as suas numerosas vantagens, tais como alta disponibilidade, elasticidade e ubiquidade. Tanto utilizadores individuais como empresas têm vindo a escolher cada vez mais estes serviços, apesar de tal implicar uma perda de privacidade e de controlo. Terceirizar dados e computações para servidores remotos implica a confiança nos seus provedores, um problema conhecido pelos utilizadores finais. Notícias recentes provam que dados guardados na Nuvem são vulneráveis a fugas de dados, tanto por parte do provedor, de atacantes, ou mesmo de programas de vigilância governamentais, expondo dados privados dos utilizadores.

Ao longo dos anos, diferentes vias têm sido seguidas na tentativa de solucionar estes problemas. Abordagens mais ingénuas envolvem guardar os dados sempre cifrados no servidor, decifrando-os apenas do lado do cliente. No entanto, tal implica uma grande sobrecarga do lado deste, tornando tais aborgadens impraticáveis. Nos últimos anos, as Cifras Simétricas Pesquisáveis (CSP) surgiram como uma nova linha de investigação, permitindo pesquisas e actualizações eficientes sobre bases de dados cifradas em servidores na Nuvem, mantendo ao mesmo tempo garantias de privacidade. Ainda assim, não obstante alguns progressos recentes, esquemas recentes ainda balanceiam eficiência, segurança e expressividade das pesquisas, limitando a sua adopção como tecnologia viável, particularmente em cenários de larga escala.

Novas tecnologias fornecendo Ambientes de Execução Isolados (AEI) podem introduzir melhorias na literatura de CSP. Estas tecnologias permitem que aplicações sejam corridas remotamente com garantias de privacidade no processador, em isolamento de outros processos potencialmente privilegiados, tais como o núcleo do sistema operativo. Exemplos proeminentes destas tecnologias são o Intel SGX e o ARM TrustZone, que têm vindo a ser disponibilizados no mercado em processadores comuns.

Nesta tese estudamos tecnologias de hardware confiável em detalhe, explorando a sua aplicação sobre o problema da computação sobre dados cifrados, focando-nos no Software Guard Extensions (SGX). Mais especificamente, estudamos a aplicação de AEIs em esquemas de CSP, melhorando as vertentes de eficiência, segurança e expressividade das pesquisas.

Desenhamos, implementamos e avaliamos três novos esquemas de CSP para diferentes tipos de pesquisas, particularmente pesquisas Booleanas sobre texto, pesquisas de imagens por conteúdo sobre repositórios das mesmas e pesquisas multimodais sobre texto e imagem. Estes esquemas permitem suportar pesquisas combinando diferentes formatos de *media* em simultâneo, tendo em vista casos de uso tais como diagnósticos médicos com suporte a privacidade e gestão de registos médicos electrónicos, ou repositórios de imagens confidenciais, que podem estar alojados em serviços provisionados pela Nuvem sem perigo de quebras de privacidade.

Palavras-chave: Cifras Simétricas Pesquisáveis; Computação na Nuvem; Privacidade; Hardware Confiável; Pesquisas Booleanas Seguras; Consulta Segura de Imagens Por Conteúdo

Contents

List of Figures

List of Tables

Listings

List of Algorithms

Glossary

ECALL Enclave Call. A call from an untrusted application into an eclave in the Intel Software Guard Extensions (SGX) architecture.

EPID Enhanced Privacy ID. A group signature scheme used to remotely attest enclaves from the Intel Software Guard Extensions (SGX) architecture.

HMAC Keyed-Hash Message Authenticated Code. Similarly to a MAC, an HMAC allows for integrity and authenticity verification of a message. It can also be used as a pseudo-random function.

MAC Message Authenticated Code. A piece of information used to attest the integrity and authenticity of a message, guaranteeing it has not been tampered.

OCALL Outside Call. A call from inside an eclave to an untrusted function in the Intel Software Guard Extensions (SGX) architecture.

Acronyms

BoVW	Bag of Visual Words.
CBIR	Content-Based Image Retrieval.
CNF	Conjunctive Normal Form.
DBMS	Database Management System.
DNF	Disjunctive Normal Form.
DPE	Distance Preserving Encodings.
EDL	Enclave Definition Language.
EPC	Enclave Protected Cache.
FHE	Fully-Homomorphic Encryption.
IaaS	Infrastructure as a Service.
IEE	Isolated Execution Environment.
IND-CKA2	Indistinguishability under Adaptive Chosen Keyword Attacks.
IND-CKA1	Indistinguishability under Non-Adaptive Chosen Keyword Attacks.
IND-OCPA	Indistinguishability under Ordered Chosen Plaintext Attacks.
IND-CPA	Indistinguishability under Chosen Plaintext Attacks.
ISR	Inverse Square Rank.
LSH	Locality-Sensitive Hashing.
mAP	Mean Average Precision.
MIE	Multimodal Indexable Encryption.
MMU	Memory Management Unit.
mOPE	Mutable Order-Preserving Encoding.

OPE	Order-Preserving Encryption.
ORAM	Oblivious RAM.
ORE	Order-Revealing Encryption.
PCR	Platform Configuration Register.
PHE	Partially-Homomorphic Encryption.
PRF	Pseudo-Random Function.
PRM	Processor Reserved Memory.
SGX	Software Guard Extensions.
SMC	Secure Monitor Call.
SSE	Searchable Symmetric Encryption.
SWHE	Somewhat-Homomorphic Encryption.
TCB	Trusted Computing Base.
TOCTOU	Time Of Check to Time Of Use.
TPM	Trusted Platform Module.
UDF	User-Defined Function.

Chapter 1

Introduction

Cloud computing has become a computing standard in recent years, and the trend to outsource data and its processing to third-party servers is expected to keep growing (Columbus, 2017b). However, by transferring sensitive and private data to the Cloud, users and organisations effectively transfer the ownership of their data to the provider (Chow et al., 2009), all assumptions about their data's security and privacy now being given solely by that provider.

1.1 Context and Motivation

In 2018 more than 96% of IT professionals reported using Cloud services (RightScale, 2018). In fact, Cloud computing presents itself with several advantages against the conventional data centre model: elasticity (the capacity to scale resources according to demand), the pay-as-you-go model, the elimination of concerns relative to maintenance, high availability, and cheap geo-replication (Armbrust et al., 2010). The growth and popularity of Cloud-based services is also evident when considering individual users; Gmail alone reported more than a billion users in 2016 (Lardinois, 2016). Consumer Cloud storage services, like iCloud or Dropbox, have also surpassed the half a billion users mark (Apple Insider, 2016; Darrow, 2016).

The growth of media sharing over the internet (Fung, 2015) has been one of the key factors for the success of Cloud computing. With mobile devices dominating Internet traffic (Titcomb, 2016), the advantages of Cloud are more evident, as it bridges the resource constraints of such devices, and allows for efficient and cheap storage and processing of data. In the future, end users will increasingly produce more data: in 2025, the world will have produced 163 zettabytes of data, a third of which will be image and video data (Reinsel et al., 2017). Moreover, the amount of sensitive data (*e.g.*, medical records and

applications) is expected to reach 15% of all data. Having large volumes of outsourced data also brings the need for supporting efficient searches, allowing users to retrieve relevant information while keeping it remotely on the Cloud.

1.2 Problem

Although Cloud computing keeps growing, its users are aware and cite security guarantees as their main concern when adopting such services (Columbus, 2017a; Meeker, 2017). These concerns are not without a cause; in recent years, several cases of privacy breaches have surfaced. Some of these are intended company policies, such as using data assumed private for advertising purposes (Rushe, 2013); other breaches may also be the result of government-ordered surveillance programs (Cook, 2016; Greenwald and MacAskill, 2013). These kind of leaks affect not only companies, but also individual users (Hough, 2010; Lewis, 2014; Turner, 2016). Particularly sensitive information, like health records, have also been the subject of several attacks (HCA News, 2018; O'Hara, 2017; Roston, 2017); with the recent growth in personal health apps usage (Khalaf, 2014; Meeker, 2017), the need for data security increases. In light of these concerns and news, Cloud services present themselves as a double-edged sword for its users: *how to leverage the advantages of the Cloud without compromising privacy and security?*

Classical approaches for securing data on untrusted servers usually relied on an *Encryption at Rest* strategy (MongoDB, 2018; MySQL, 2018). Data was stored encrypted on the server, and all update and search operations required data to be downloaded, decrypted, and computed at the client. Such strategies, however, are not only inefficient, but also require high network and client-side computational overheads.

Throughout the years, techniques such as cryptographic file systems tried to allow some computations to be made over (possibly remote) encrypted data (Goh et al., 2003; Wright et al., 2003). Yet, these approaches are limited to traditional file I/O, and, therefore, do not allow operations such as searches over that data.

The issue can be described as that of *computing over encrypted data*. If data stored remotely can remain encrypted and queried without performance penalties, then Cloud services can be used without loss of privacy. Homomorphic encryption (Gentry, 2009; Rivest et al., 1978b), for example, allows for arbitrary computations over encrypted data – it supports searching with high query expressiveness; moreover, it also provides strong security guarantees. However, current solutions still have low performance and are, therefore, far from practical. Other proposals, like property-preserving encryption (Pandey and Rouselakis, 2012), provide different sets of operations over encrypted data, but usually leak more information to the Cloud provider than desirable.

The field of Searchable Symmetric Encryption (SSE) (Curtmola et al., 2006; Song et al., 2000) appears as a novel and more interesting approach, as it allows for some computations over encrypted data, while being practical and providing better security guarantees than property-preserving approaches. The field has been a hot research topic

in the last years, with proposals trying to address security issues (Stefanov et al., 2014) and improving performance (Cash et al., 2014; Ferreira et al., 2017). Query expressiveness has also been an issue of note, as newer solutions address not only queries over text data, but also over images and multimodal data (*i.e.* data with multiple media formats simultaneously) (Ferreira et al., 2015; Ferreira et al., 2017).

All the aforementioned solutions are software-based. They usually consider a honest-but-curious adversary, which observes all data and computations on the Cloud server without interfering; such model implies that sensitive data can never be decrypted on the server. In property-preserving and SSE schemes, this can imply several rounds of communication between clients and servers, incurring in network and computational overheads. However, novel trusted hardware solutions based on remote attestation, such as Intel SGX (Hoekstra et al., 2013; McKeen et al., 2013) and ARM TrustZone (ARM, 2009), appear as way to execute programs in untrusted environments, while keeping full confidentiality and integrity guarantees. The formal definition of an IEE (Barbosa et al., 2016), whereby programs can be executed on remote servers with security guarantees, facilitates a more comprehensive security analysis of systems comprising hardware-based solutions. Approaches where SSE schemes include trusted hardware are still few (Fuhry et al., 2017), and by leveraging a trusted environment in a Cloud setting, one can reduce the overhead of network communications, as computations formerly done by the client can now be executed remotely.

1.3 Objectives

Current SSE schemes are still a trade-off between three dimensions: security, query expressiveness, and performance. While some provide better security guarantees, they usually relax performance guarantees (Bost et al., 2017); some schemes provide better usability and query expressiveness, but may compromise security (Kamara et al., 2012).

In this thesis we design, implement and evaluate novel SSE schemes using trusted hardware solutions. We tackle current SSE limitations with this new approach, namely by providing better security guarantees (through the use of secure hardware), improving query expressiveness (by supporting Boolean queries over text data, similarity queries over images, and queries over multimodal data), which allows us to provide richer types of operations, and offering an efficient and practical solution (by securely leveraging Cloud resources). More broadly, we aim at answering the following question:

How can we improve SSE solutions to provide better security guarantees, while also being practical and efficient?

Our objective is to design SSE schemes that are practical and can be validated experimentally. For this purpose, we implemented all contributions, assessing their performance and security. Our final objective is to provide a multimodal framework for efficient and secure simultaneous text and image querying.

1.4 Main Contributions

We can summarise the contributions of this thesis as follows:

- **I – BISEN** A SSE scheme for Boolean ranked querying, supporting queries with an arbitrary number of conjunctions, disjunctions and negations, over a database of text documents.

- **II – VISEN** A SSE scheme for Content-Based Image Retrieval (CBIR) over image databases.

- **III – MISEN** A SSE scheme gathering the previous solutions in a multimodal framework, supporting privacy-enhanced queries over different types of data.

All three contributions are based on a common Cloud architecture for secure execution of our SSE protocols. We also developed a practical Framework to help abstract Intel SGX and provide an IEE-like interface, which we used to implement our schemes. All of our prototypes are made available as open-source software.

1.5 Publications

Two publications have been made in the context of this thesis:

- *BISEN: Efficient Boolean Searchable Symmetric Encryption with Verifiability and Minimal Leakage.* Technical Report.
  Bernardo Ferreira, Bernardo Portela, Tiago Oliveira, Guilherme Borges, Henrique Domingos, and João Leitão.
  Cryptology ePrint Archive, Report 2018/588. 2018.
  Available on `https://eprint.iacr.org/2018/588`.

- *Pesquisa Booleana Cifrada usando Hardware Confiável.*
  Guilherme Borges, João Leitão, Henrique Domingos, and Bernardo Ferreira.
  Proceedings of the 10th Simpósio de Informática (INForum'18). Coimbra, Portugal. 2018.
  Available on `https://novasys.di.fct.unl.pt/~gb/papers/inforum18.pdf`.

1.6 Thesis Organisation

This thesis is organised as follows:

Chapter 2 analyses the state-of-the-art and the related work relevant for our contributions, mainly in the areas of Searchable Symmetric Encryption (SSE) and trusted hardware.

Chapter 3 presents and analyses our three schemes and their protocols, providing their security analysis and discussing possible improvements.

Chapter 4 describes our approach for the implementation of our schemes, together with a Framework developed for generic Isolated Execution Environment (IEE) computation.

Chapter 5 presents the experimental evaluation for our schemes.

Chapter 6 concludes the thesis and presents future work directions.

Chapter 2

Related Work

To provide context for this thesis we intend to analyse and discuss current SSE schemes, together with novel trusted hardware approaches to allow computation outsourcing to remote providers. Our analysis of the state-of-the-art will be divided between four main areas: computations over encrypted data, with an emphasis on Searchable Symmetric Encryption (SSE), existing trusted hardware solutions, techniques for information retrieval, and an overview of existing solutions which apply principles of the previous areas.

This chapter is organised as follows: Section 2.1 presents different cryptographic mechanisms and approaches to perform computations over encrypted data; on Section 2.2 we present some information retrieval techniques and concepts relevant for this thesis; Section 2.3 analyses and discusses the state-of-the-art on searchable encryption; in Section 2.4 we discuss different approaches on trusted hardware; Section 2.5 presents existing prototypes and frameworks that use principles and techniques referred to in the previous sections; finally, in Section 2.6 we discuss and summarise the state-of-the-art presented during the chapter.

2.1 Computing over Encrypted Data

Computing over encrypted data has been a topic of research since 1978 (Rivest et al., 1978b). With the recent advent of Cloud-based technologies, it has become an intense field of research, due to the security concerns raised by outsourcing sensitive data to third-party servers. In this section we present two different categories of encryption schemes particularly useful in the context of untrusted servers: property-preserving and homomorphic encryption.

2.1.1 Property-Preserving Encryption

Property-preserving encryption schemes are encryption schemes whose ciphertext retains a given property or set of properties of their plaintext, and thus such properties can be publicly inferred (Pandey and Rouselakis, 2012). The two most usual relations those schemes preserve are either determinism or order; allowing arbitrary computations can be achieved by composing different schemes or through more expensive primitives like Fully-Homomorphic Encryption (FHE) (see Section 2.1.2).

Deterministic Encryption Deterministic properties are shown by any deterministic encryption scheme (Bellare et al., 2008), *e.g.* AES in ECB mode – with the same plaintext and key, the same ciphertext is generated. This allows equality queries over encrypted data, which can be particularly useful in encrypted databases (Popa et al., 2011). First definitions on deterministic encryption were introduced by Bellare et al. (2007), who proposed asymmetric encryption-based schemes, and noted that, by having the same plaintext words generating the same ciphertexts, word frequency would be leaked to the adversary. Nevertheless, if word entropy is high, *i.e.* no words are repeated throughout the datastore, deterministic encryption does not leak such information.

Bellare et al. (2007) also proposed an application of their schemes in untrusted storage (Efficient Searchable Encryption – ESE), allowing for efficient (sub-linear) performance by using auxiliary tree-based data structures.

Order-Preserving Encryption Order-Preserving Encryption (OPE) was first proposed by Agrawal et al. (2004) and retains the numeric ordering of plaintexts when encrypted[1], *i.e.* a function f is order-preserving if, for any two numeric inputs a, b,

$$a < b \iff f(a) < f(b)$$

and an encryption scheme is order-preserving if its encryption algorithm is an order-preserving function (Boldyreva et al., 2009).

Boldyreva et al. (2009) were the first to define and formally analyse OPE schemes, proving that they did not provide Indistinguishability under Chosen Plaintext Attacks (IND-CPA), as an adversary can establish order between two generated ciphertexts. The same work also proposed the notion of Indistinguishability under Ordered Chosen Plaintext Attacks (IND-OCPA), defining that minimal leakage was the ordering of ciphertexts. Later, Boldyreva and Chenette (2011) improved upon the previous definition, as it also leaked the distance between plaintexts and their high-order bits. Popa et al. (2013) achieved ideal security (IND-OCPA) with Mutable Order-Preserving Encoding (mOPE), a scheme requiring existing ciphertexts to be mutated upon new inserts on a datastore; the main insight of mOPE is that practical OPE needs both statefulness and mutability of ciphertexts.

[1] In fact, OPE is also inherently deterministic, but in this section we will refer to it as a special case of determinism.

Florian Kerschbaum (2015) proposed an OPE scheme that preserved order but hided equality, thus hiding keyword frequency and increasing security. The scheme, however, introduced a margin of error in some queries, apart from requiring client storage.

A related approach is that of Order-Revealing Encryption (ORE), proposed by Boneh et al. (2015). Conversely to OPE, where ciphertext values are numeric, and thus comparable directly, in ORE the values are comparable through a known function which takes two ciphertexts as input, outputting its plaintext ordering, also achieving IND-OCPA. ORE is at least as secure as OPE (Chenette et al., 2016); the fact that order is only revealed when the function is passed to the server further increases such guarantees.

2.1.2 Homomorphic Encryption

Encryption schemes are homomorphic if they preserve additive or multiplicative properties of the plaintext while performing such operations over encrypted data. The first homomorphic encryption schemes were proposed by Rivest et al. (1978b), and are based on the multiplicative property of the RSA algorithm (Rivest et al., 1978a).

Homomorphic encryption schemes can be divided into Fully-Homomorphic Encryption (FHE) schemes, which preserve both additive and multiplicative properties of the plaintext in its ciphertext (thus allowing for any arbitrary computation over encrypted data), and Partially-Homomorphic Encryption (PHE) schemes, which only preserve a given property of the original plaintext.

Partially-Homomorphic Encryption Some encryption schemes can be considered partially-homomorphic, as their ciphertext preserves either additive or multiplicative properties of the plaintext. Unpadded RSA (Rivest et al., 1978a) and ElGamal (ElGamal, 1985) preserve multiplicative properties, while Paillier (Paillier, 1999) is additively homomorphic.

The Paillier algorithm is a public-key scheme that shows homomorphic additive properties. To perform encryption over a value v, a public key (n, g) and a random number r are generated, such that $0 \leq v < n$ and $0 \leq r < n$; the ciphertext is generated from the plaintext[2] (considering a public key $pk = (n, g)$ and a private key sk[3]) by

$$Enc(v, pk) = g^v \cdot r^n \bmod n^2$$

The addition property is shown by

$$Dec(Enc(v_1, pk) \cdot Enc(v_2, pk) \bmod n^2) = v_1 + v_2 \bmod n$$

Since a multiplication operation can be performed as a set of additions, the algorithm also shows the multiplicative property with

$$Dec(Enc(v_1, pk)^{v_2} \bmod n^2, sk) = v_1 \cdot v_2 \bmod n$$

[2]The previous restrictions limit Paillier-encrypted messages to a message space of a maximum of n.

[3]The private key is generated together with the public key, whose parameters and generation algorithm we omit for brevity.

On the other hand, the ElGamal (ElGamal, 1985) cryptosystem, for example, preserves multiplicative properties, and is based on the Diffie-Hellman key exchange algorithm. Like RSA, ElGamal shows the property

$$Dec(Enc(v_1, pk) \cdot Enc(v_2, pk), sk) = v_1 \cdot v_2$$

Somewhat-Homomorphic Encryption Somewhat-Homomorphic Encryption (SWHE) schemes allow for arbitrary computations up to a certain depth, *e.g.* by allowing arbitrary additions and one multiplication, of which the construction proposed by Boneh et al. (2005), based on bilinear maps, is an example.

Fully-Homomorphic Encryption The first practical FHE scheme was proposed in 2009 by Craig Gentry (2009) and, by supporting both additive and multiplicative properties, it allows for any arbitrary computation over encrypted data, as all computations can be reduced to Boolean and arithmetic operations. By leveraging SWHE schemes (Boneh et al., 2005), Craig Gentry proposed a scheme capable of evaluating its own decryption function, which allows for recursive embedding of ciphertexts, each corresponding to an operation over encrypted data. Implementations still proved unpractical (Gentry and Halevi, 2011); more recently, schemes for AES encryption using FHE have been proposed (Gentry and Halevi, 2011) and, although being more practical, are still much slower than their unencrypted counterparts.

2.1.3 Oblivious RAM

Oblivious RAM (ORAM), introduced by Goldreich (1987) and Goldreich and Ostrovsky (1996), is based on the concept of oblivious Turing machines – a Turing machine is oblivious if, for two different inputs of the same length, its tape movements are indistinguishable from each other. As such, ORAM conceals memory access patterns by shuffling and continuously re-encrypting data on memory, fitting the untrusted server model, where the server provider can probe data buses on their machines to perform statistical and inference attacks.

Recent approaches to ORAM design have been motivated by the emergence of Cloud computing, and either try to improve ORAM's performance (Stefanov et al., 2012; Stefanov et al., 2013), or consider subsets of memory – data structures – aiming at providing full security only for such structures (Wang et al., 2014). Stefanov et al. (2013) proposed Path ORAM, which sees memory as data blocks, organised in a binary tree; when a block is accessed, its position in memory is changed, and thus an attacker can only infer how many accesses to memory are made, but not which parts are more accessed. However, to this date, all proposed ORAM schemes are still far from being practical in real-world usage.

2.1.4 Discussion

On the one hand, property-preserving encryption is usually practical and relies on widely-known algorithms, allowing for some computations over encrypted data, like equality (deterministic schemes) or aggregation operations (OPE schemes), which are particularly useful in database systems (Popa et al., 2011). Yet, by their own design, they inherently leak information about their data. When used on a datastore, these schemes are also susceptible to inference attacks over time – by performing operations and adding records, they allow an adversary to progressively learn information about the respective plaintext data. On the other hand, homomorphic encryption schemes, while providing probabilistic (or IND-CPA) security[4], are still far from practical (FHE) or provide little usability (PHE), preserving only a single arithmetic property. Work on ORAM is still mainly theoretical and, although a promising path for full security in Cloud applications, it still incurs in performance penalties of several orders of magnitude (Wang et al., 2014). Therefore, the mentioned schemes are usually a trade-off between performance, security, and query expressiveness.

2.2 Information Retrieval Techniques

Information retrieval techniques prove useful when searching over large quantities of data, particularly on remote datastores, where the cost of data transferral is high, *e.g.* due to latency. The most basic technique is exact-match searching, and can be applied to several media types, in particular text. Even so, more advanced techniques might prove more useful: in very large datasets, exact-match queries can return a large subset of documents; metrics like term frequency (number of occurrences of a given keyword in a document) prove useful to attribute scores to query results, and can be applied through ranking functions, like *TF–IDF* (Jones, 1972), *BM25* (Manning et al., 2008, Section 11.4.3), and the Vector Space Model (Salton et al., 1975). These techniques allow results to be ordered by ranking, returning only the most relevant ones. In this section we will discuss text retrieval schemes (Section 2.2.1), image retrieval schemes and complementing clustering techniques (Section 2.2.2), and different metrics for information retrieval schemes (Section 2.2.3), finalising with a discussion of these different techniques (Section 2.2.4).

2.2.1 Text Retrieval

Text data is a prime example for information retrieval schemes (Manning et al., 2008, Section 1). As such, to improve query performance over large text datastores, different indexing and querying techniques have been proposed. A particular structure is the inverted index, which consists of a dictionary mapping keywords to documents identifiers

[4]Homomorphic encryption schemes are malleable and, therefore, do not provide security against chosen ciphertext attacks (Katz and Lindell, 2007, p. 104).

containing them (Zobel and Moffat, 2006). With this index, metrics like document frequency can be easily inferred by counting the document ids of a keyword, while metrics like term frequency might need auxiliary data structures. More advanced improvements on index efficiency and query result relevancy can be achieved by preprocessing documents, which include the removal of stop words (common but irrelevant words like *and* or *the*), text stemming (storing only the root of the word in the index) (Bassil, 2012), and others.

2.2.2 Image Retrieval and Feature Clustering

Different approaches for image retrieval have been proposed. One of the most interesting ones is Content-Based Image Retrieval (CBIR) (Kato, 1992), as it allows querying by example (similarity), a technique close to those used by many search engines nowadays (Jing et al., 2015; Thomas, 2017, pp. 36–37). However, other such methods exist, like searching by text annotations, added either manually or automatically (Jeon et al., 2003).

To index and store images on a datastore, one can consider either global or local features (Mikolajczyk and Tuytelaars, 2009, pp. 939–943), which are stored in data structures known as feature vectors. Global features produce a single vector for image characterisation, an example of which are colour histograms (Swain and Ballard, 1991). Local features apply for a specific part of the image, and are usually defined by measuring changes in colour, intensity, or texture of an image.

Feature vector extraction from an image usually yields a large quantity of highly dimensional data (algorithms such as SURF (Bay et al., 2008) or SIFT (Lowe, 2004) produce vectors with 64 and 128 dimensions, respectively); searching over such data becomes impractical. One of the first approaches to allow for efficient image data searching was the Bag of Visual Words (BoVW) (Nistér and Stewénius, 2006), which constructed a tree of feature vectors based on a given image training dataset. Similar feature vectors extracted from the training dataset would be grouped into the same tree nodes, forming groups of similar feature vectors. Image vectors would be added to the most similar node in the tree, and searches would be done similarly by comparison with existing groups. By classifying feature vectors into groups, or *bags*, image data can then be treated similarly to text data, where each *bag* is akin to a keyword, enabling images to be stored by using indexes. This technique can then be combined with approaches like the inverted index (Zobel and Moffat, 2006) to associate features to image documents.

Construction of the feature vector tree can more broadly be described as a clustering technique. Several image retrieval schemes which use such techniques have been proposed (Ferreira et al., 2015; Jegou et al., 2008; Liu et al., 2014b), usually resorting to unsupervised machine learning algorithms, such as k-means (Lloyd, 1982; MacQueen, 1967). In the remainder of the section we discuss two approaches for the k-means algorithm, and alternative clustering techniques, such as Locality-Sensitive Hashing (LSH) or binarisation.

Traditional K-means The traditional k-means algorithm (Lloyd, 1982) takes a training dataset as input, and outputs a codebook of k centroids, *i.e.* a set of clustered feature vectors. When feature vectors are added to the datastore they are compared to the codebook centroids, and are clustered into the closest one; thus, each vector can be seen as a keyword from a small group of k keywords.

K-means is composed of two phases: seeding and updating (Manning et al., 2008, Section 16.4). In the seeding phase, cluster centroids are chosen randomly from the dataset; during the update phase, clusters are adjusted to better fit the training dataset vectors, and the latter are adjusted into new clusters if needed. At the end of the algorithm, distances between each vector and their cluster centre should be as minimal as possible.

Online K-means The online version of the k-means algorithm (MacQueen, 1967) works similarly to the traditional one, but only visiting each training vector once. This approach is interesting for resource-constrained settings, or Big Data applications and scenarios, as vectors do not need to be stored in memory or disc. However, this method is more sensitive to the order the vectors are presented in, and thus can be less precise. Some improvements to this approach have been proposed recently, and apply techniques such as block processing – perform traditional k-means of smaller blocks, and online k-means of the resulting centroids (Aaron et al., 2014), adding clusters as needed during each iteration (Bagirov et al., 2011), varying k dynamically to improve precision (Aaron et al., 2014), or weighting clusters according to their relevance in each iteration (Bagirov et al., 2011).

Locality-Sensitive Hashing Locality-Sensitive Hashing (LSH) is a family of hash functions used to convert multi-dimensional data into a scalar value. Contrarily to cryptographic hash functions, LSH's objective is to produce the same hash value for similar vectors; collisions of similar values are therefore desirable. Given its property of grouping similar vectors together, LSH can be used as a clustering algorithm.

Voronoi-based LSH (Loi et al., 2013), for example, works by generating k random feature vectors following a normal distribution; these vectors are then used as cluster centroids. This technique has the main advantage of not requiring a training step, which can be computationally expensive.

2.2.3 Result Scoring and Evaluation

Improving search results usually involves document ranking functions, such as *TF–IDF* (Jones, 1972). This function combines *Term Frequency* with *Inverse Document Frequency*, taking into account the frequency of a keyword not only on the document, but on the whole datastore. Thus, it privileges occurrences of rare keywords and relegates more popular ones. *Term Frequency* is usually the frequency of a searched keyword in a document,

while *Inverse Document Frequency* for a keyword w is obtained by calculating

$$IDF(w) = \log \frac{N}{f}$$

where N is the total number of documents in the datastore, and f the number of documents containing w (*i.e.*, its datastore document frequency).

To assess and evaluate information retrieval systems some measures have been proposed, of which we highlight precision and recall (Teufel, 2007, Chapter 3). Precision refers to the fraction of documents, retrieved by a query, that are relevant to that search. Recall, on the other hand, is the fraction of relevant documents for a given query that are effectively retrieved from the complete datastore. A single metric can be obtained with Mean Average Precision (mAP) (Manning et al., 2008, p. 147), which averages precision over a set of queries. While these metrics are not relevant for exact-match searches, they can prove a valuable tool for the assessment of systems when performing ranked and similarity queries.

To account for multimodal systems, and combine different types of ranking functions, rank fusion metrics have been proposed. These apply after each ranking engine is executed, and combines their scores to produce a unified one. One of these metric is Inverse Square Rank (ISR) (Mourão et al., 2013), which is defined as

$$ISR(d) = f * \sum_{e=1}^{f} \frac{1}{s(e,d)^2}$$

where f is the document frequency of d across all engine results and $s(e,d)$ is the score of document d on an engine e.

2.2.4 Discussion

Nowadays, the field of information retrieval developed many different techniques and schemes for efficient querying over large datastores, with an emphasis on the relevancy of responses, and support for different media types. Many of these techniques do not take security concerns into account, which may curb its adoption in real-world scenarios. Some SSE schemes have already adopted information retrieval techniques, albeit not always together with performance concerns. By merging techniques from both areas, better, more practical and secure schemes, can be developed.

2.3 Searchable Symmetric Encryption

Searchable Symmetric Encryption (SSE) aims to provide efficient searches on untrusted servers, while preserving security guarantees and having significant query expressiveness. It was first proposed by Song et al. (2000), and considered the untrusted server model. This server was a text document storage (each document seen as a set of keywords), and

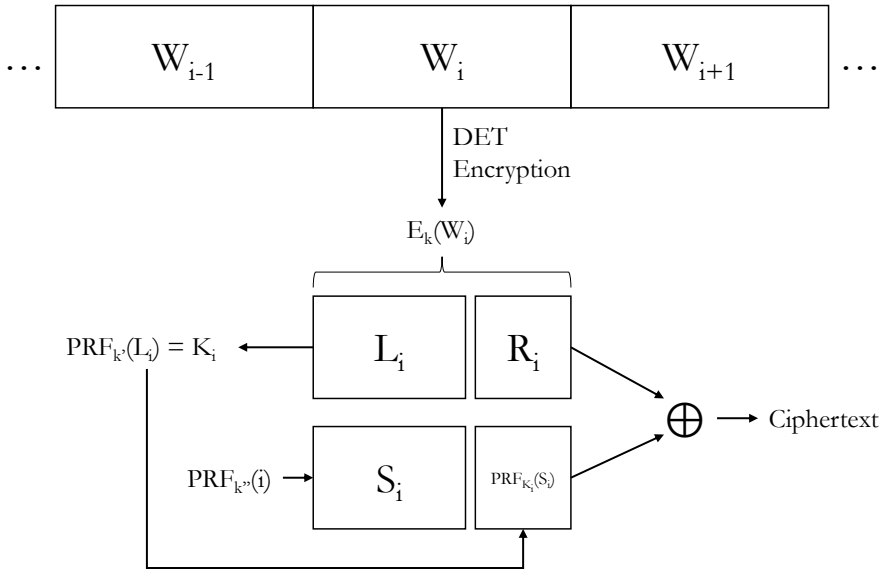


Figure 2.1: SSE encryption algorithm of Song et al. (2000).

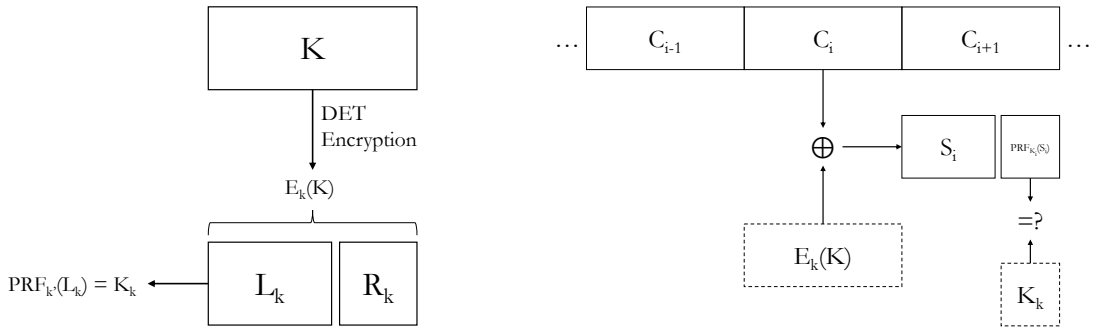
the scheme achieved linear performance in regards to storage size[5]. Significant improvements over SSE constructions were proposed by Curtmola et al. (2006), while in recent years dynamic SSE was proposed (Kamara et al., 2012). In this section we will analyse the main contributions to this field of research, while detailing practical applications of SSE for Boolean text queries in Section 2.3.1, for ranked retrieval in Section 2.3.2, and for image data in Section 2.3.3, discussing the state-of-the-art in Section 2.3.4.

Origins The original scheme for SSE by Song et al. (2000) provided algorithms for encryption and search over text data. In Figure 2.1 we present the complete version of the encryption algorithm: each word W_i (at position i) of the document is **encrypted** separately with a deterministic scheme and key k; then, the resulting ciphertext $E_k(W_i)$ is split in two parts, L_i and R_i. At the same time, a Pseudo-Random Function (PRF) acting as a stream cipher generates a random string of bytes S_i with key k'', dependant on the position of the keyword. L_i from the ciphertext is used to generate a key K_i, used in a PRF that takes S_i as input, and whose output is appended to it. Then, the ciphertext $L_i|R_i$ is XORed with $S_i|PRF_{K_i}(S_i)$.

To perform a **search**, as shown in Figure 2.2, the client starts by encrypting the desired keyword in a manner similar to the encryption algorithm, obtaining both $E_k(K)$ and K_k, which are then sent to the server[6]. The server then iterates over all encrypted keywords, performing a XOR with the received $E_k(K)$; this recovers the intermediate step of the encryption phase. Then, the right side of the XORed ciphertext is compared with K_k sent from the client; if it is equal, then the searched keyword was found.

[5] In the remainder of this section, we will refer to performance as the relation between search latency and storage size, except where otherwise noted.

[6] These client tokens were later designated as *trapdoors* in the literature.



a Client side.

b Server side. Dashed boxes represent information sent from the client.

Figure 2.2: SSE search algorithm of Song et al. (2000).

This scheme guarantees IND-CPA, which means that an adversary cannot distinguish two entries in the storage, even by generating new entries and comparing them to pre-existing ones (Bösch et al., 2014) – encrypted entries do not appear deterministic, as they contain a source of randomness. The scheme leaks the position of keywords in the document, thus being susceptible to statistical attacks, especially as more searches are performed, which reveals keyword popularity.

Indexing and Security Guarantees Additional improvements and security guarantees in SSE were introduced by Curtmola et al. (2006), who proposed the use of indexes to achieve better performance (sub-linear searches) in SSE schemes. Furthermore, Curtmola et al. also provided the notion of Indistinguishability under Non-Adaptive and Adaptive Chosen Keyword Attacks (IND-CKA1 and IND-CKA2, respectively). Both guarantee that only search patterns and its results are leaked, which is considered a strong security definition (Bösch et al., 2014); non-adaptive adversaries have no knowledge of previous searches, while adaptive ones do, the latter model corresponding to real-world applications of SSE (Curtmola et al., 2006).

Albeit preserving the privacy of stored documents, SSE schemes can still leak *search* and *access patterns* (Liu et al., 2014a). The former identifies whether a query was performed before, while the latter which document identifiers are returned by a given query. As time progresses, this leakage will accumulate, progressively revealing more about the datastore's content.

Dynamic SSE and State-of-the-Art Up to 2012, all proposed schemes with IND-CKA2 guarantees only supported static document storages; therefore no documents could be added, updated or removed without rebuilding the whole storage. Kamara et al. (2012) proposed a dynamic SSE scheme that allows these operations by adding new data structures that track such operations efficiently, while still keeping the same strong security guarantees from Curtmola et al. (2006). However, it also introduced the notion of *update*

leakage, by which an adversary can infer which inserted keywords already existed in the datastore, and their respective document ids.

Cash et al. (2014) proposed a new scheme, based on an encrypted dictionary of pairs $(label, documentID)$, which is similar to the concept of an inverted index from the information retrieval field (see Section 2.2.1). Labels are generated by combining a keyword and a counter, stored by the client, indicating the number of documents containing that keyword. This scheme was the first to have no *update leakage* and also presented sub-linear performance.

Most of these schemes are designed for a single user scenario. Although some schemes allow for a multi-user setting (Cao et al., 2014), these usually require a single user to generate trapdoors. Popa et al. (2014) proposed a scheme for multiple-keyword searches for multiple users, yet, it was based on elliptic curve cryptography, which carries significant performance penalties.

The notion of *forward* and *backward privacy* in SSE schemes is also important and was introduced by Stefanov et al. (2014), who also proposed a scheme were the former was addressed. *Backward privacy* requires searches to leak no information about previously deleted keywords, while *forward privacy* entails that adding a document does not reveal whether any of its keywords have been searched before. *Forward privacy* was formally defined by Raphael Bost (2016) and further improved with definitions for *backward privacy* in Bost et al. (2017) (where the authors proposed different schemes achieving both types of privacy), as previous SSE literature only tackled the issue informally.

2.3.1 Boolean Queries

Most SSE schemes are designed for single-keyword searches over text data (Cash et al., 2014; Curtmola et al., 2006; Song et al., 2000). The first scheme considering multiple-keyword queries was proposed by Golle et al. (2004), and supported Boolean conjunctive queries with linear performance. Cash et al. (2013) proposed a scheme supporting both conjunctive and disjunctive queries, with sub-linear and linear performances, respectively.

More recently, Kamara and Moataz (2017) proposed a solution supporting Conjunctive Normal Form (CNF) queries with sub-linear performance regarding database size. Their main construction, *IEX-2LEV*, is composed of encrypted dictionaries and multi-maps. The scheme contains a global multi-map, mapping every existing keyword w to its document identifiers, and local multi-maps for each of these keywords – an auxiliary dictionary is used to map each of those keywords to their local multi-map. The local multi-map for w then maps all keywords v that co-occur with w to the respective document identifiers. Therefore, query performance becomes quadratic regarding its own size (number of keywords per query), while it also requires quadratic storage in regards to unique keywords. The current scheme trivially supports disjunctive queries; to perform conjunctions, thus enabling CNF queries, several composite searches are performed, and

the resulting sets of documents are intersected as needed.

2.3.2 Ranked Searching

Most SSE schemes are still focused on exact matches; on very-large datasets, however, ranked searches provide a tool for obtaining only the top-ranking (and, therefore, more meaningful) documents (Swaminathan et al., 2007). Wang et al. (2012) proposed the first SSE scheme to allow ranked searches on single keywords; their scheme leveraged OPE to order documents based on the frequency of searched keywords per document. Ferreira and Domingos (2013) proposed a dynamic scheme to index text data using inverted indexes; the scheme used homomorphic encryption to store scoring data with each keyword occurrence; searches would send encrypted ranking results to the client, which then decrypted and sorted documents. Cao et al. (2014) proposed the first multi-keyword ranked scheme, however, it carried large performance penalties for the client and leaked search and frequency patterns. This scheme also required an essentially static datastore, being unable to support dynamic updates, deletes and inserts.

2.3.3 Images

Privacy-preservation on image datastores has been considered in recent literature. Lu et al. (2009) were the first to propose a privacy-preserving CBIR scheme; it used the Bag of Visual Words (BoVW) method with an index for features that had to be generated and encrypted client-side (with schemes such as OPE), while searches would leverage properties preserved by such schemes. While other initial proposals focused on privacy-preservation for feature extraction algorithms, via techniques such as homomorphic encryption (Hsu et al., 2012), the first practical solution for a privacy-preserving CBIR scheme on untrusted servers (*IES–CBIR*) was proposed by Ferreira et al. (2015). The scheme works by separating images into different components; in IES–CBIR colour and texture information. As texture is more relevant than colour in image recognition (Wang et al., 2001), texture data is encrypted with a probabilistic scheme, which has stronger security guarantees than the deterministic encryption used for colour. In the server, data is indexed according to colour information. To perform searches, users generate trapdoors based on a querying image; the server extracts its colour information, performing a comparison against the existing datastore.

Xia et al. (2016) proposed a scheme using feature vector extraction and LSH to cluster image features; for searches it considered an index of feature vectors for linear comparison, plus an extra LSH-based index for faster lookup.

More recently, Distance Preserving Encodings (DPE) have been proposed (Ferreira et al., 2017) to allow data encoding with privacy guarantees for both sparse (text) and dense (image, video, and sound) media types. DPE for dense media types is based on the extraction of feature vectors from the plaintext object, and encoding them in a privacy-preserving way; the resulting encodings preserve Hamming distance up to a threshold

Schemes	Access	Search	Update	Frequency
Curtmola et al. (2006)	x	x	n/a	n/a
Cao et al. (2014)	x	x	n/a	x
Cash et al. (2014)	x	x	–	n/a
Ferreira et al. (2017)	x	x	x	x
(Homomorphic) Ferreira et al. (2018)	x	x	–	–

Table 2.1: Leakage comparison of the SSE state-of-the-art. 'x' indicates patterns leaked by a scheme, whereas 'n/a' indicates that such leakage does not apply to the scheme. '–' indicates patterns not leaked.

given as parameter, *i.e.* distances bigger than such threshold are not preserved and, in essence, information leakage can be controlled by the user when encrypting.

For sparse media types DPE is optimised – text is limited to a given subset of words and, as such, searches can be limited to equality comparisons, for which a threshold of 0 is used. Achieving proximity searches for text can be obtained through techniques like stemming, thus obtaining results not only for exact-matches, but also for words with the same root.

As DPEs allow for privacy-preserving encodings, heavier operations, like dataset training, can be outsourced to an untrusted server, through a Multimodal Indexable Encryption (MIE) framework. MIE performs feature vector extraction and encoding on the client-side, then training and indexing data on the server-side.

MuSE (Ferreira et al., 2018) uses SSE techniques to reduce leakage in multimodal schemes, hiding document *update patterns* revealed in MIE; the authors further propose using homomorphic encryption to hide document and keyword frequency.

2.3.4 Discussion

Current SSE schemes already provide some security guarantees; yet, many do not contemplate high query expressiveness, or have a high client-side overhead. The topic of security has been approached with definitions such as *backward* and *forward privacy*. Nonetheless, efficiently providing both *backward* and *forward privacy* without weakening one in detriment of the other is still an open research field, which no solution fully addresses yet (Bost et al., 2017). Moreover, the issue of leakage – minimising the amount of information exposed to an untrusted server – is still being improved, as guaranteeing minimal leakage in SSE implies no leakage apart from *access leakage*. The issue of verifiability – addressing active adversaries – also remains largely unaddressed, as Cloud providers are assumed honest-but-curious, *i.e.* do not tamper with data or computations. Table 2.1 presents a comparison of leakage from schemes discussed in this section. While *update leakage* is not present in more recent schemes, *search patterns* are still leaked by all current schemes. Furthermore, hiding *access patterns* to memory is an issue that cannot be solved solely by SSE techniques, requiring other methods such as ORAM.

Some attacks on SSE have been proposed recently by exploring the inherent leakage many schemes have. Cash et al. (2015) analysed the problem by performing statistical attacks where the document set was varyingly known to the attacker; knowing the full document set allowed to recover more than 80% of queries, whereas knowing only 70% of the document set lowered the recovery rate to almost 0%.

To solve some of these problems, many SSE schemes imply client-side computations, which can also increase communication overheads, restricting their practical applications, and largely limiting the computational power of Cloud servers. Ideally, SSE schemes should be balanced between query expressiveness, performance, and security. With this in mind, one may leverage newer approaches, like trusted hardware, to achieve both better performance and stronger security guarantees than those given by traditional approaches.

2.4 Trusted Hardware

According to Radu Sion (2009), trusted hardware denotes hardware that is certified to meet a given set of security requirements, usually defined as counter-point from an adversarial model. Trusted hardware can provide privacy, integrity or attestation guarantees over the instructions it executes, even on a remote machine, *e.g.* a Cloud server. More broadly, the recent notion of trusted computing was proposed by Santos et al. (2009), directed at providing a platform to outsource computations to Cloud servers with confidential guarantees.

Nonetheless, using hardware to provide isolation and improve security guarantees is an idea proposed since, at least, 1978 (Rivest et al., 1978b). Robert Best (1981) submitted a patent for a microprocessor to execute encrypted programs. This microprocessor decrypted a program as it was executed, allowing for its source and data to be always encrypted outside of it. In the following years, others works were proposed in the same field; *ABYSS* (White, 1987) was the first solution providing an architecture whereby an application is divided into unprotected and protected parts, thus resembling the architecture of Intel SGX (Section 2.4.3.1).

More recently, different approaches to trusted hardware have been presented, each with different target guarantees. In this section we present some of these different approaches, such as the secure coprocessor (Section 2.4.1), the Trusted Platform Module (TPM) (Section 2.4.2), and then focus on the recent notion of an Isolated Execution Environment (IEE) (Section 2.4.3), and existing implementations fitting its model, namely Intel SGX (Section 2.4.3.1) and Arm TrustZone (Section 2.4.3.2).

2.4.1 Secure Coprocessor

A secure coprocessor is a hardware module responsible for cryptographic and sensitive computations, being designed to be added onto standard computers. Bennet Yee (1994)

defines a secure coprocessor as a hardware module requiring internal processing, a dedicated ROM, and secure storage. Secure coprocessors must also be physically secure, *i.e.* they must be tamper-proof, eliminating all sensitive data when a physical intrusion is detected.

One of the most popular implementations of a secure coprocessor was the IBM 4758 (Dyer et al., 2001). The 4758 design lays on the following principles: tamper resistance, the existence of a source of randomness, a layered architecture to isolate different applications with different privileges, authentication capabilities, and persistent storage. In fact, many of these design principles are present in more modern hardware solutions, like the TPM.

2.4.2 Trusted Platform Module

A Trusted Platform Module (TPM) (Trusted Computing Group, 2009; Trusted Computing Group, 2015) is a specification for a hardware module aimed at providing a Trusted Computing Base (TCB) easily applicable on commodity hardware. TPM chips provide three main different kinds of service: software attestation, boot authentication, and encryption (Stallings and Brown, 2014, pp. 485–489), the latter essentially provided by an internal secure coprocessor. A TPM is considered to be trustworthy and tamper-proof; by attesting software running on a machine, it further extends the trust base onto such software. Apart from the mentioned services, a TPM also contains internal volatile and non-volatile memory, the latter used to store encryption keys and records used for boot authentication. In the remainder of this section we present and discuss the three main aspects of the TPM.

Boot Authentication The TPM chip provides an authenticated boot, which verifies that a machine's operating system has not been tampered with. When booting, the OS code is loaded from the disc onto volatile memory; the TPM produces hashes of software modules being loaded (including BIOS firmware and the bootloader), storing those hashes in internal registers – Platform Configuration Registers (PCRs) – and comparing them with predetermined ones stored in its non-volatile memory. If a hash of the code being loaded does not match the expected one, the boot process is aborted and the user is notified.

Software Attestation Similarly to boot attestation, a TPM can verify the loading sequence of a program into memory. Before loading it, the TPM produces a hash of the program's code, storing it in a PCR. The user, which can be local or remote, possesses the expected PCR values. When needed, they can request a *Quote* from the TPM by sending a nonce and required PCR values to the TPM. The TPM produces a signed proof, the *Quote*, consisting of the requested PCR values combined with the nonce and signed with the TPM's private key; the user can then verify that *Quote*, inferring whether the program's PCR values correspond to the expected ones.

Secure Coprocessor A TPM chip also includes a minimal secure coprocessor, which is able to perform cryptographic operations, like random (or pseudo-random) number generation, encryption, and hashing. This can be used to seal data stored in normal unprotected storage; the keys are stored inside the TPM, which guarantees that data can only be decrypted on that machine, and the keys cannot be stolen.

Discussion TPM devices provide attestation guarantees on commodity hardware, allowing to certify that software running on a computer can be trusted, without the need to modify such software. These guarantees, albeit strong, only refer to the loading phase of the program: they do not detect code injection at run time. To prevent such attacks, a user might have to restart the program at a given interval, so the loading and attestation process can be performed again. Therefore, an adversary can still exploit kernel faults and inject code with an attack known as Time Of Check to Time Of Use (TOCTOU). This particular problem was addressed by Bratus et al. (2008), which proposed dynamically updating PCR values when the memory it measures is changed.

Although the authenticated boot mechanism can attest the integrity of an operating system, it also follows that no update to the OS can be done without implying resetting all boot attestation records.

By storing keys internally, the TPM does not offer a solution to recover sealed documents in case of permanent damage; data either is lost or would have to be backed up in another machine unencrypted.

Some attacks to the TPM have also surfaced in the literature. Kursawe et al. (2005) proposed a passive attack by analysing the unencrypted data bus between the TPM and the CPU. Kauer (2007) and Sparks (2007) analyse an active attack, called the *TPM Reset Attack*, which forces a reset of the module, possibly causing erroneous PCR readings.

2.4.3 Isolated Execution Environment

An Isolated Execution Environment (IEE), as defined by Barbosa et al. (2016), is an abstraction that describes a machine capable of executing a given program P in an isolated environment, *i.e.* whose output is determined by the initial state of P and a set of defined inputs given into that environment. The IEE offers full isolation from other programs running on the same machine – both on data and computations, consequently providing privacy against untrusted programs on that machine.

To allow outsourcing of programs running on IEEs, the notion of *attested computing* is introduced to assure a user that P is running untampered on a remote IEE. Attestation is defined as a set of three algorithms: *Compile*, *Attest* and *Verify*. *Compile* corresponds to the initial bootstrap and attestation of P; *Attest* is ran on the remote machine and interacts with the program running on the IEE to produce an attestation proof; *Verify* is ran by the user and checks the validity of the output from *Attest*.

```
enclave {
    include "secure_stdio.h"

    trusted {
        public void ecall_malloc_free(void);
        public void ecall_sgx_cpuid([out] int cpuinfo[4], int leaf);
    };
    untrusted {
        void ocall_print_string([in, string] const char* str);
        int ocall_read(int fd, [out, size=size] void* buf, unsigned int size);
    };
};
```

Listing 2.1: Sample of an EDL file, based on the Intel SGX SDK code samples (Intel, 2018a).

Finally, to create a secure communication channel between a user and their remote program running on the IEE, a key exchange algorithm is performed, which ensures that all subsequent communications preserve integrity, confidentiality, and authentication guarantees.

In practice, Intel SGX and ARM TrustZone can be adapted to fit the notion of IEEs; we analyse both technologies in the remainder of the section.

2.4.3.1 Intel Software Guard Extensions (SGX)

Intel Software Guard Extensions (SGX) (Anati et al., 2013; Hoekstra et al., 2013; McKeen et al., 2013) consist of a set of instructions available on most currently available Intel CPUs. These instructions allow the programmer to define IEEs, known as enclaves. Enclaves can be seen as containers where protected code can be run with confidentiality and integrity guarantees (McKeen et al., 2013) provided by hardware.

An application designed under the SGX software model is divided into an untrusted and a trusted component – the latter being the enclave (Hoekstra et al., 2013). When the untrusted component of the application requests the creation of an enclave, the CPU allocates a region of the memory – up to a practical limit of 128 MB (Intel, 2018b) – available only to that specific enclave, and therefore inaccessible from any other application running in the same machine, even from the untrusted component requesting the enclave and the system kernel. This memory is kept encrypted in the RAM, being only decrypted by the CPU when requested by the enclave.

Enclave code can be called upon by the untrusted code via special instructions (ECALLs). While on an enclave, no access to common system libraries, like `stdio.h` and `stdlib.h`, is provided[7], as the SGX trust base excludes operating systems (McKeen et al., 2013). Therefore, access to networking or disc I/O is made possible through the use of OCALLs, which allow calling functions in the untrusted component from inside the enclave. When

[7] Some popular libraries are now provided as OCALL abstractions by the SGX SDK (Intel, 2017, pp. 39 – 40). However, these are special implementations made by Intel for the SDK, which implies most libraries are still unavailable directly on the enclave.

an OCALL is performed, the enclave is temporarily exited, the untrusted function is executed, and the enclave is re-entered.

The definition of ECALLs and OCALLs in an application is done through a specification file, known as the Enclave Definition Language (EDL) file. An example of such file can be found in Listing 2.1. The file contains a header and two blocks, `trusted` and `untrusted`, corresponding, respectively, to ECALLs and OCALLs. The header can define C-style inclusions to trusted libraries (as the ones provided by the SGX SDK). Function definitions in the `trusted` and `untrusted` blocks also mimic C-style function headers, with the addition of special attributes on pointer parameters. These attributes, defined in square brackets, describe which security checks have to be made in order to pass pointers to and from the enclave.

Apart from the confidentiality and integrity guarantees provided by the enclave model, SGX also provides tools for remote attestation, which, similarly to the corresponding TPM feature, allow a user to certify that a remote enclave is running correctly and untampered.

Programming Model The CPU manages a region of memory called Processor Reserved Memory (PRM), isolated from every component of the system (including the OS and the hypervisor) apart from the CPU itself; the PRM holds all enclaves' data structures, along with their code and data (McKeen et al., 2013). The PRM also contains the Enclave Protected Cache (EPC), a set of protected memory pages that can be assigned to enclaves. To perform enclave operations, SGX's instruction set provides eighteen different instructions, of which we present and summarise the most relevant in Table 2.2.

These instructions exist to perform different enclave operations, data integrity checks, attestation, sealing, and also debugging. An enclave is created by calling the `EECREATE` instruction, which allocates the needed Enclave Protected Cache (EPC) pages and initialises SGX's enclave data structures. Further pages can be added to the enclave via the `EADD` instruction. To end the initialisation process, `EINIT` is called, and the enclave can then be run.

Calls into an enclave function (ECALLs) are performed by the `EENTER` instruction, enabling the CPU's enclave mode; to return, `EEXIT` is called. When an exception occurs, `AEX` is called, returning from the enclave into the untrusted program. All enclave data, including the CPU registers, is stored securely in the PRM; returning to the enclave is then made possible by the `ERESUME` instruction. SGX's instruction set also contains debug instructions to allow reading and writing into an enclave during development, being unavailable on a production enclave.

Page Eviction Memory inside the PRM is limited. In spite of that, to allow multiple applications concurrently, the instruction set supports swapping (evicting) pages between the PRM and untrusted memory (McKeen et al., 2013). When a page is to be evicted, the instructions `EBLOCK` and `EWB` are called, respectively blocking writes and encrypting the

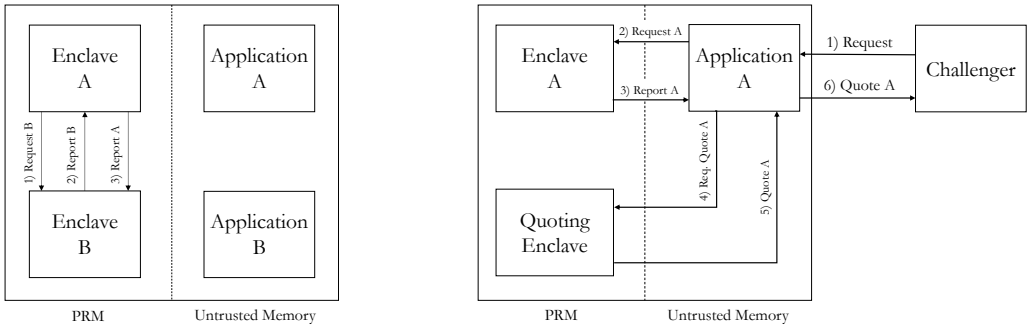
Category	Instruction	Description
Initialisation	EECREATE	Declare an enclave, its size, and initialise data structures.
	EADD	Add a 4KB Enclave Protected Cache (EPC) page to the enclave.
	EREMOVE	Remove an EPC page from the enclave.
	EINIT	Finalise the enclave initialisation, enabling its execution.
Entering and Exiting	EENTER	Enter an enclave via an ECALL.
	EEXIT	Exit the enclave.
	AEX	Asynchronous exit from the enclave, saving the CPU state internally.
	ERESUME	Resume the execution of the enclave, restoring previously stored state.
Paging Management	EBLOCK	Block an enclave page, preparing it for eviction from protected memory.
	EWB	Evict a page from protected memory.
	ELDB/U	Reload an evicted page into protected memory.
Enclave Security	EREPORT	Produce a *Report structure*, which contains information about an enclave and the hardware it is running on.
	EGETKEY	Provide an enclave with different keys used internally, like Report or Sealing keys.

Table 2.2: Subset of the instruction set from Intel SGX. From Anati et al. (2013) and McKeen et al. (2013).

page, so that it can be stored in untrusted memory. The evicted page contains integrity proofs (a MAC of the contents and a version number) to guarantee it was not tampered with after reloading. To reload a page, via ELDB/U, its contents are copied into the PRM again, the page decrypted, and security checks performed, ensuring its contents have not been tampered and the page's version is the last one, thus preventing replaying-like attacks.

Attestation SGX also provides ways to attest a running enclave, guaranteeing to the user that it has not been tampered with. When an enclave is initialised, a digest of its code and data is produced and stored internally in a special register (MRENCLAVE), a process known as measurement (Anati et al., 2013). Attestation can then be performed, both between local enclaves and between an enclave and a remote user (Figure 2.3); the latter being the case of an enclave running on the Cloud.

To perform **local attestation** (see Figure 2.3a), an enclave *A* performs an attestation request to enclave *B* (*step 1*), *B* then calls the EREPORT instruction, producing a *Report Structure* containing *B*'s identity, its attributes, and information about the hardware it



a Local attestation. b Remote attestation.

Figure 2.3: Intel SGX attestation mechanism for local and remote settings. Based on Anati et al. (2013).

is running on, further producing a MAC of that structure via a symmetric Report Key (owned by the CPU, and given to the enclave by the EGETKEY instruction). This structure is sent to enclave *A* (*step 2*); *A* can then use the same Report Key from the CPU to produce their *Report Structure* of itself, comparing the received MAC from *B* with the one generated in *A*; if they are running on the same platform, it follows that the MACs should be equal. Finally, *A* reciprocates the attestation process and produces its own *Report Structure* to send to *B* (*step 3*).

To perform **remote attestation** (see Figure 2.3b), a challenger starts by issuing a request to the untrusted application *A* (*step 1*), which acts as an intermediary for communication. The application *A* requests a *Report Structure* from its enclave, which creates it as in the local attestation process (*steps 2 and 3*). Finally, the application sends the *Report Structure* to a special enclave (Quoting Enclave) (*step 4*). This enclave signs the structure with an EPID key, producing a *Quote* and returning it to the challenger (*steps 5 and 6*). The EPID key is an asymmetric key owned by the CPU; the remote user performing the attestation must have a public key certificate of that EPID, which they use to verify the *Quote* from SGX.

Sealing Another of the features of SGX is data sealing, which can be seen as a form of encryption. As the enclave size is limited, this feature allows for data to be stored encrypted outside the enclave boundaries, while preserving the same security guarantees given by it. A Sealing Key can be requested via the EGETKEY instruction, and is unique to that enclave. Further iterations of the same enclave will get the same Sealing Key, for the enclaves will have the same identity, *i.e.* its attributes, thus allowing for long-term storage of data.

2.4.3.2 ARM TrustZone

ARM TrustZone considers a similar architecture to that of Intel SGX by providing the notions of *Normal* and *Secure Worlds*. Conversely with SGX, where enclaves are seen as

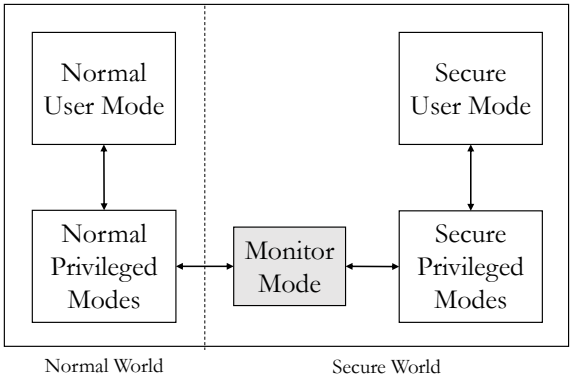


Figure 2.4: ARM TrustZone architecture. Each world is effectively seen as its own virtual core, although both exist in the same physical core. Based on ARM (2009).

regions of memory, in TrustZone the *Secure World* corresponds to a virtual CPU core; a physical core is seen as two virtual cores, an untrusted and a trusted one (ARM, 2009). Transitions between *Normal* and *Secure Worlds* are assured by a monitor component (Figure 2.4), which saves all registers and protects sensitive memory when switching from one *World* into another.

The *Secure World* gives the guarantees of an IEE by provisioning an execution environment fully isolated from other, untrusted, applications.

Programming model Switching between worlds implies a new instruction, Secure Monitor Call (SMC), which is supported by the existence of a new CPU privileged mode, the monitor mode. When transitioning between modes, the CPU assures all sensitive data is flushed from shared registers (both *Worlds* share the same physical core), in a similar fashion to that of SGX's instructions `EEXIT` and `AEX`.

Designing applications for TrustZone is a fundamentally different task from that of SGX, as no fixed programming model is imposed by ARM, such as the enclave model of SGX. The simplest one can be regarded as similar to enclaves, and consists of synchronous calls into a library inside the *Secure World*; as such, *Secure World* calls become slaves from *Normal World* applications. As ARM states (ARM, 2009), this model is often enough for most applications. On the other end of the spectrum, TrustZone allows for the use of a secure OS kernel running entirely inside the *Secure World*; this approach allows for multiple applications running in parallel inside it, but implies the design of such solution by the programmer. The level of complexity needed and the way to handle the *Secure World* software stack is, therefore, a task of the programmer.

Hardware Architecture TrustZone's architecture, of which a high-level overview is presented in Figure 2.4, consists of two virtual cores, one for each *World*, and several duplicated hardware modules, which provide efficient physical isolation. Memory management, for example, is assured by the existence of two physical Memory Management

Units (MMUs), one for each virtual core, allowing for totally independent memory mappings. One of the advantages of this strategy is removing the need for MMU flushes when switching *Worlds*.

Caches for each *World* are also handled differently, although sharing the same physical housing. Cache lines of TrustZone CPUs are marked with an additional bit indicating the security state of that line, thus enabling the CPU to segregate accesses from both *Worlds*, removing the need for cache flushes when switching *Worlds*; moreover, cache lines from one *World* can be evicted by the other without prejudice of security.

Finally, TrustZone also provides a segregated bus (AMBA3 APB) to secure peripheral access, namely to I/O devices, guaranteeing that a secure peripheral can not be accessed from the *Normal World*. TrustZone also provides a mechanism for interrupt handling in the *Secure World* – exceptions are treated inside it, which is not the case of Intel SGX.

2.4.3.3 Discussion

By considering both software and hardware to be untrusted (apart from the CPU), SGX eliminates many kinds of attackers from the literature (Costan and Devadas, 2016, Sections 3 and 6.6). However, it provides no protection against side-channel attacks, which recent literature has been exploring (Seo et al., 2017; Shih et al., 2017; Xu et al., 2015). These kind of attacks usually explore page faults and SGX's handling mechanism, which relies on transferring control to the operating system. More recently, the Spectre attack (Kocher et al., 2018) has been found to also affect SGX (O'Keeffe et al., 2018), by exploiting conditional branch speculative execution. These kind of attacks, however, rely on bugs found over optimisations offered by Intel CPUs – they are not based on flaws of the system model and architecture. Therefore, and as they can be patched either via software or hardware replacements, they do not challenge SGX's security guarantees.

TrustZone's notion of IEE, materialised as the *Secure World*, provides a more general-purpose notion to the programmer in comparison to SGX's enclaves, which only allow a synchronous function calling model. The inclusion of peripheral bus communication into TrustZone's trust base offers further tools for the programmer, albeit at the cost of programming complexity and availability of trusted firmware. By including devices from outside the CPU into TrustZone's trust base, however, applications that depend on third-party firmware become endangered if vulnerabilities are found in that firmware. Intel SGX's policy on the inclusion of external library applications, although restrictive, offers a smaller, more secure, trust base.

Although not a recent concept, leveraging hardware to provide security guarantees has become more accessible to users as such resources become integrated with commodity hardware, more specifically CPUs. Hence, these solutions will prove more practical to Cloud providers than previous ones, as they require no direct handling of hardware to be available; servers can be supplied as-is by the provider, the setup and use of the IEE becoming a responsibility of the user.

2.5 Data Repositories and Frameworks for Secure Computation

The concepts presented in the previous sections, particularly Sections 2.1 and 2.4, have been applied in different research prototypes and software. In Section 2.5.1 we present some research prototypes designed to enhance pre-existing database solutions by provide privacy guarantees, leveraging on property-preserving or homomorphic schemes in a transparent way to its users. In Section 2.5.2 we present frameworks for trustworthy computations based on trusted hardware, and that either try to integrate existing solutions into the trusted setting (Haven or VC3), or provide novel SSE schemes based on hardware (HardIDX).

2.5.1 Secure Data Repositories

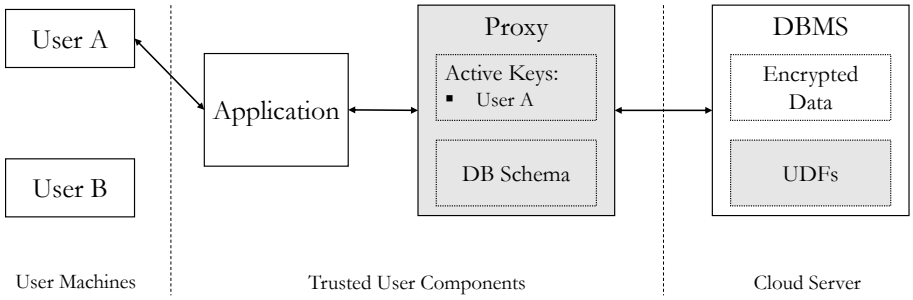


Figure 2.5: Simplified view of CryptDB's architecture. The proxy and application are assumed to be trustworthy at most times. CryptDB's components are shown shaded. User A is logged in, while User B is not. Based on Popa et al. (2011).

CryptDB Popa et al. (2011) were the first to approach privacy-preserving data repositories that leveraged on schemes like OPE or homomorphic encryption, basing their CryptDB solution over MySQL databases. Using different property-preserving schemes, it guarantees total privacy of data stored on a remote server, while also providing efficient searches on data. To support join operations, CryptDB also introduced two novel schemes, allowing for equality and order based-joins.

CryptDB aims to provide a transparent SQL interface to the user by using a trusted middleware (proxy), while keeping all data in the untrusted server always encrypted. As shown by Figure 2.5, CryptDB only adds a proxy and User-Defined Functions (UDFs) to an already-existing application setup. The proxy is responsible for all encryption and decryption operations and handles communication between the Database Management System (DBMS) and the application. CryptDB uses PHE and property-preserving schemes to encrypt the fields of an SQL database; a given plaintext field is converted into several encrypted fields on the remote database, each with a different scheme, thus incurring in storage overheads. The mapping from plaintext into encrypted columns is stored in the proxy (as the DB schema), and is handled dynamically as tables are created or updated. To support schemes like Paillier (Paillier, 1999), which involve performing multiplications to achieve additions over ciphertext data, UDFs are used by the DBMS.

Supported operations depend on the encryption schemes used; complex mathematical operations (like sine), for example, are not supported by any existing property-preserving encryption scheme; as such, not all SQL queries are supported. CryptDB's adversary model is based on an untrusted Cloud provider – an adversary can snoop on all the user's data, albeit not tampering with it. This attacker acts on the *Cloud Server* of Figure 2.5, and is known as honest-but-curious. The system also provides partial protection against active attacks on both the *Trusted User Components* or the *Cloud Server*: by having each user encrypt their data with their own key, an attacker can only gain access to secrets from logged-in users, while others have their data safe, as decryption keys are only stored in the proxy for active users.

TrustedDB Bajaj and Sion (2011) proposed TrustedDB, which relies on trusted hardware (a secure coprocessor) on an untrusted server. TrustedDB provides a fully-fledged SQL interface, which allows client-side components to remain unchanged, conversely to CryptDB's approach of adding a new component on the client-side. Security guarantees of TrustedDB are based on that of its secure coprocessor, as data is never shown to the untrusted server, and handled entirely in that coprocessor.

Cipherbase Arasu et al. (2013) employed the same principles from CryptDB and TrustedDB in Cipherbase – transparency to users issuing queries and total privacy of remotely-stored data. Cipherbase leverages on trusted hardware (FPGAs) to achieve a fully-fledged database system. Queries that cannot be resolved by using PHE and property-preserving schemes are resolved in the trusted hardware module, maintaining data confidentiality from a Cloud provider. Instead of adding a middleware between client and server, Cipherbase requires a trusted machine located near the untrusted server, similarly to TrustedDB; in this machine, all data encryption and decryption operations are performed, using the untrusted server to perform queries over encrypted data.

Google Encrypted BigQuery Encrypted BigQuery (Google, 2016) is a modification of the client for Google's BigQuery (Google, 2010) Infrastructure as a Service (IaaS) solution. This solution provides the same guarantees of CryptDB, while being adapted for larger dataset processing. Yet, by being based on the same kind of schemes from CryptDB, it also suffers from the same drawbacks: only some operations are supported, and they may carry additional performance and storage overheads.

Arx More recently, Arx (Poddar et al., 2016) improved upon previous solutions by providing stronger encryption schemes. Proposing an architecture similar to that of CryptDB, Arx uses two proxies, a trusted one on the client-side, and an untrusted one on the server-side. Built upon MongoDB, Arx uses secure indices to provide the same properties as property-preserving schemes, though with IND-CPA guarantees, leaking only the number of elements in the database, but not information like order.

2.5.2 Frameworks for Trustworthy Computation

Haven The Haven framework (Baumann et al., 2014) aims to provide *shielded execution* guarantees for applications running on untrusted Cloud providers; such guarantees are similar to that of IEEs in regards to process isolation and confidentiality. Based on Intel SGX (Hoekstra et al., 2013; McKeen et al., 2013), Haven aims to supply a trusted platform for full, unmodified applications to run on, unlike typical SGX applications, whose design needs to consider which parts need to be executed trustworthily. By executing unmodified applications in the enclave, Haven needs to support OS calls, for which a library with an OS API is provided (LibOS). LibOS, in turn, is supported by a shield module, which provides kernel functionalities inside the enclave. With this architecture, Haven does not reduce the trust base, but tackles the problem of Iago attacks (Checkoway and Shacham, 2013), which are directed at system calls. Nonetheless, the overhead of the Haven stack implies a loss of performance up to 50% in the emulated SGX used; and no hardware implementation was tested.

VC3 VC3 (Schuster et al., 2015) is a framework for Intel SGX whose goal is to execute trustworthy Hadoop MapReduce jobs on untrusted Cloud providers. Therefore, VC3 defines a protocol for distributed MapReduce jobs with attestation, key exchange and verification steps, as all data that is passed between enclaves needs to be secured. VC3, conversely to Haven, reduces the trust base to the traditional SGX model, where only the CPU is trusted, and adds the libraries needed for MapReduce to the enclave.

Ohrimenko et al. (2016) Ohrimenko et al. (2016) proposed a framework to perform several different machine-learning algorithms with SGX, in a privacy-preserving multi-party computation setting. The authors propose several data-oblivious primitives, which include assignment and array access operations. These primitives are then used to guarantee protection against side-channel attacks on memory, which have been found to affect SGX. The machine-learning algorithms analysed are also adapted to be used in the trusted multi-party setting, and experimental results on hardware show that SGX data-oblivious primitives do not impact computations significantly on most cases.

EnclaveDB EnclaveDB (Priebe et al., 2018) is a database management system leveraging an Intel SGX enclave for sensitive data storage. The system considers a client–enclave secure channel, with query decryption and processing totally done in the enclave, thus ensuring data confidentiality and integrity. However, to fully support a DBMS inside the enclave, the model relies a theoretical version of Intel SGX with virtually unlimited memory.

HardIDX HardIDX (Fuhry et al., 2017) is a SSE scheme leveraging on trusted hardware (Intel SGX) to efficiently and securely perform range queries. The scheme encodes order

information in a B+ tree, which is stored inside the trusted enclave (alternatively, the tree can be stored in the untrusted memory if it grows to surpass the enclave's memory). To the best of our knowledge, HardIDX is the first SSE scheme to depend on trusted hardware for its design.

2.5.3 Discussion

Solutions like CryptDB and Arx are convenient to users and existing applications, as they support unmodified database management systems, respectively MySQL and MongoDB. Current and popular applications are supported with little to no changes, as common queries are simple and can be resolved by simple encryption schemes (Popa et al., 2011). Solutions like Cipherbase and TrustedDB, on the other hand, provide full fledgeness, but require specialised trusted hardware that Cloud providers rarely supply.

With the advent of IEEs in commodity CPUs (Section 2.4.3), new approaches on these systems can be taken – secure coprocessors and FPGAs required specialised programming interfaces (Bajaj and Sion, 2011), becoming unpractical and costly to develop, while newer approaches, like SGX, have similar programming interfaces to that of current commodity systems. The current trend is to adapt previous solutions to the trusted hardware on the Cloud setting, yet, existing proposals are either too broad (Baumann et al., 2014) and carry performance penalties, or are designed with a specific application in mind (Schuster et al., 2015); they provide, however, a useful insight into the adoption of newer IEE-based technologies.

2.6 Summary and Discussion

In recent years, especially with the rise of Cloud computing, the notion of *computing over encrypted data* has become an intense field of research. Although the concept was first proposed in 1978, only the last decade has seen most of its advancements. On the one hand, the advent of property-preserving encryption allowed for an early insight into the topic, specifically concerns over its query expressiveness, performance, and security. On the other hand, homomorphic encryption schemes, although based on a different approach, also provide malleable ciphertext. Yet, currently both approaches mainly lack practicality; property-preserving encryption is usually a trade-off between security and performance; fully homomorphic encryption is secure and has high query expressiveness, but is still far from practical.

The field of information retrieval is rich in techniques and methods for searches over large datastores, although without security guarantees. The field of Searchable Symmetric Encryption (SSE) can provide useful techniques and insights that can be complemented with information retrieval solutions. Various SSE schemes have been proposed recently; these achieve a trade-off between the three metrics (security, performance, and query expressiveness) we have been referring to. Initially, concerns over performance dominated

such schemes (Song et al., 2000); security guarantees are also increasingly taken into account, with the first formal definitions of Indistinguishability under Adaptive Chosen Keyword Attacks in 2006 (Curtmola et al., 2006), and new definitions of *forward* and *backward privacy* in 2014 (Stefanov et al., 2014) and 2017 (Bost et al., 2017), respectively. Interest over query expressiveness is also growing, as recent schemes proposed richer queries, including Boolean text (Kamara and Moataz, 2017) or image similarity data (Ferreira et al., 2015; Ferreira et al., 2017), by leveraging concepts from information retrieval schemes. While a hot topic of research, many proposed SSE schemes still lack full security guarantees, and simultaneously tackling different privacy concerns has been proven a difficult goal in the current literature.

Still, many of these schemes do not consider the full possibilities of the Cloud's computational power and data capabilities. With the concept of trusted hardware, new possibilities can be explored by outsourcing some client computations, which need to be private, onto Cloud servers. Trusted hardware has also been a field of research for many years (since 1978). However, and even though many different approaches have been made in the field, both academic and industrial, wide-adoption was never achieved. More recently, with the formal definition of an Isolated Execution Environment (IEE) (Barbosa et al., 2016), and the inclusion of such hardware in commodity CPUs (like Intel SGX), the field has gained more traction, with prototypes like Haven (Baumann et al., 2014) or VC3 (Schuster et al., 2015) exploring its potential.

So far, solutions that combine concepts from these areas are still close-to-none (Fuhry et al., 2017). As the Cloud provides more and more services, the opportunity to leverage them with privacy guarantees and more trustability assumptions (with the required minimisation of dependable TCBs), is now bigger than ever and, therefore, exploring the field presents itself as a new and interesting path.

Chapter 3

Protocols for Isolated Searchable Encryption

In this chapter we introduce three schemes and their respective protocols for isolated searchable encryption: one for text data, one for image data, and one for multimodal data. Our schemes combine traditional SSE techniques with trusted hardware, particularly an Isolated Execution Environment (IEE) component. This component allows us to securely execute computations on untrusted machines, like a Cloud environment, thus leveraging its advantages, such as high computational power and large persistent storage.

State-of-the-art SSE usually finds a trade-off between compromising some security guarantees (to enable more efficient server-side execution), allowing for the leakage of some operation patterns, or choosing to execute computations client-side (preserving security but incurring in high network and computational overhead). Using an IEE allows us to efficiently outsource computations to otherwise untrusted Cloud servers, while still preserving strong security guarantees. Furthermore, by outsourcing computations to the Cloud, we can consider thin client devices, such as mobile devices.

Our two first schemes, BISEN and VISEN, allow for search over encrypted text and image data, respectively. Our third scheme, MISEN, combines text and image data into multimodal queries. We start this chapter by introducing the common system model and architecture for our schemes (Section 3.1), present two relevant use cases for our solution (Section 3.2), provide the necessary definitions and tools used to define our solutions (Section 3.3). We conclude the chapter by presenting and analysing our schemes, and their respective protocols, for Boolean querying over text data (Section 3.4), Content-Based Image Retrieval (CBIR) (Section 3.5), and multimodal data (Section 3.6).

3.1 Architecture and System Model

The architecture for our solution is presented in Figure 3.1. Our protocols consider a single-client model interacting with an IEE-enabled Cloud server[1]. The Cloud server resources are divided into trusted and untrusted; the former being an IEE, the latter being composed of a regular Cloud server (a Hypervisor or Operating System) and a data storage service with large volatile and persistent storage capacity. The main functionalities of the components are as follows.

- ***Client***: creates request messages for the *IEE*, pre-processing data as needed.
- ***IEE***: executes computations with confidentiality, authentication, and integrity guarantees on a remote environment.
- ***Storage Service***: stores large amounts of encrypted data, as an index, to be used by the *IEE* with a key-value store-like interface, *i.e.* a database index.
- ***Server***: initialises the *IEE* and acts as a network proxy, mediating *Client-IEE* communications, as well as between the latter and the *Storage Service*.

We consider the *Client* to be any kind of device, from low-end mobile ones to normal desktop computers. In the Cloud server we consider the *IEE* to be resource-constrained, and thus we make no assumptions regarding its memory and storage capabilities, as some implementations offer less resources than others. Our untrusted components (*Server* and *Storage Service*) are considered fully-fledged, with no specific limitations on their resources. In particular, the *Storage Service* has access to large memory (RAM) and storage (disc) resources, while the *Server* has no limitations on its processing power[2], considerations which allow us to leverage the full power of the Cloud.

Our schemes are based on indexes for efficient search over large document datastores. Therefore, and similarly to state-of-the-art SSE, our schemes only store indexes which allow users to efficiently obtain document identifiers relevant to their queries; original documents can be stored in traditional datastores, encrypted with standard symmetric encryption algorithms, which users can then retrieve based on the identifiers returned by our schemes.

The Cloud components in Figure 3.1 can be deployed in a single machine – with enough resources to support all components; they can also be separated into two different machines running on the same data centre (*i.e.*, *Server* and *IEE* running in one, and the *Storage Service* in another), with our only requirement being that they have network communication. Moreover, each of the three components can be deployed in a different

[1] This model can be extended for multi-client support through the addition, for example, of a key management service. However, we find this extension to be orthogonal to the focus of this thesis and leave it for future work.

[2] While our schemes only require the *Server* to be a proxy, our model would allow it to execute computations if needed.

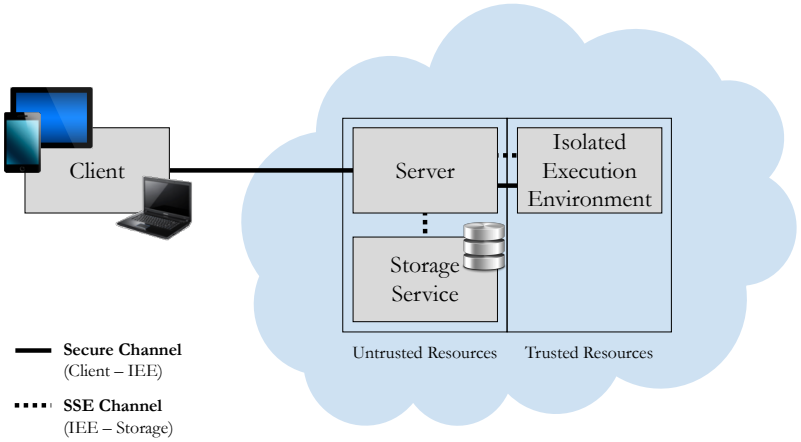


Figure 3.1: System architecture. The Cloud server is composed of trusted and untrusted resources. Through the untrusted resources, the *Client* establishes a secure channel to the *IEE*, represented by the solid line. The *IEE* interacts with the *Storage Service* through a SSE channel (a TCP unsecured channel where messages use SSE techniques to preserve security guarantees), represented by the dotted line.

machine[3]. Finally, the *IEE* can brought to the client-side – an option that ultimately achieves the conventional approach of SSE.

Communication between client devices and the *IEE* is performed through a secure channel abstraction with privacy, integrity, and authentication guarantees, instantiated through an attestation-based key exchange step described in Section 3.3. As an *IEE* has no direct communication to the outside world, its communication is mediated by the untrusted *Server*, a fact that does not imply loss of security guarantees – the underlying secure channel provides attestation and authentication for the *Client* and *IEE*. Moreover, as our *IEE* assumes very little from a memory perspective, we expand its resources with cryptographically-secured accesses to untrusted resources to store data (*Storage Service*), which can be verified when the *IEE* requests it through SSE techniques.

Adversary Model Regarding the adversary model, in our solution we will address a fully-malicious adversary, which is capable of performing incorrect operations or malicious actions, including both passive and active attacks. Passive attacks involve snooping on data, either in transit, when processed at the *Server* and *Storage Service*, or at rest; active attacks involve tampering with such data and computations. The considered threats are particularly focused on breaking confidentiality and integrity assumptions of users running critical computations involving searching, processing, and data access. By preventing against both passive and active attacks, our solution preserves full privacy and integrity guarantees, including the topology of threats usually related to the honest-but-curious model, further extended by countermeasures avoiding the injection of malicious

[3]Some IEE implementations may require an additional request forwarder to be in the same machine as the *IEE*. In such models, the *Server* component would be an entry-point for the data centre, while the *IEE* would also have a request forwarder associated.

behaviour in the Cloud's untrusted components, an enhancement that is not provided by the previous related work approaches. For clarification, we are not focused on attacks against the availability of computing and storage services, or any related Denial-of-Service attacks. Such attacks are very hard to counter, since the Cloud server and its provider control the whole infrastructure; these attacks could potentially be thwarted by increasing redundancy and replicating data through multiple servers to achieve fault-tolerance (Bessani et al., 2013), a research vector we leave as future work.

Considering our system model and architectural assumptions, our trust base includes the *Client* machine and the Cloud server's *IEE*, namely the specific processing involved in dispatching operations behind the API provided in the security surface of available IEE services, excluding all the other internal server processing resources (such as the *Storage Service*), as well as all network communications.

3.2 Use Cases

In this section we look at two possible use cases where our contributions can be of interest: personal health records and email databases, as both imply manipulation of sensitive data, while also being popular (Khalaf, 2014) and the target of recent attacks (O'Hara, 2017; Roston, 2017).

Personal Health Records Personal Health Records consist of services, like HealthVault (Microsoft, 2018), offered by Cloud providers, where users can store multiple informations about their health – these typically include medical records combining both text and image data. Users may also connect to mobile phones and wearable devices that continuously produce and upload data about them to the Cloud. These services can also be used by medical doctors, facilitating the access to patients data. Diagnosis can be improved by searching for symptom keywords and similar images on a datastore. SSE schemes can be leveraged to provide efficient searches and preserve patient confidentiality. Due to different kinds of data used, SSE schemes that provide high usability are particularly interesting.

Email Databases In this use case, users or companies outsource their email datastores to a Cloud-based storage. To retain privacy while also being able to perform searches over email data, SSE schemes with rich querying options are useful. However, Cloud email storage can be easily targeted by file-injection attacks (Zhang et al., 2016), a significant problem if the datastore is continuously updated. Therefore, to fully secure confidential email data, it is crucial to improve the security guarantees offered by SSE schemes used in this scenario, as we propose with this work.

3.3 Definitions and Tools

We now present the needed definitions and tools used to better describe our protocols. These can be grouped into *cryptographic primitives*, which provide the necessary tools to maintain our secure guarantees, *communication primitives*, which allow for inter-process and network communication, and *component primitives*, which dictate how components can interact with each other. Finally, we describe the common *initialisation procedure* of the *IEE*, needed to establish a secure channel and authenticate the *IEE* in our three protocols. In the pseudo-code, we consider the existence of data structures such as lists, tuples, dictionaries, and sets; we assume the elements in these structures are ordered by insertion unless otherwise noted.

Cryptographic Primitives As cryptographic primitives we use a Pseudo-Random Function (PRF) F and an authenticated symmetric cipher Θ. Both contain a key generation procedure *gen*(); F generates a pseudo-random transformation with *run*(key, message), and Θ encrypts and decrypts inputs with *enc*(key, message) and *dec*(key, message), respectively. We define a security parameter λ of a fixed-size in bits to reason about the security of these primitives.

Communication Primitives Our communication primitives define how *Client-IEE* communication is performed. To preserve confidentiality, integrity, and authentication guarantees of data in transit between *Client* and *IEE*, these components communicate via a secure channel. This channel lays on a standard transport layer protocol (*e.g.*, TCP), represented by `Transport` – with calls *send*(destination, message) and *recv*(message). Furthermore, to ensure message security, we use the `SecureMsg` primitives – with calls *enc*(message) and *dec*(message); these primitives encrypt and decrypt messages with an authenticated symmetric cipher, giving integrity, confidentiality, and authentication guarantees, by including a MAC with the respective ciphertext.

By combining `Transport` and `SecureMsg` primitives, we are able to establish the secure channel described in Section 3.1. Calling these primitives, therefore, ensures not only the transport of messages through the network, but also their respective encryption and decryption with full security guarantees.

Component Primitives The *IEE* component can only be interacted with through the *Server* (a restriction found in practice), which uses **IEE** primitives *init*() and *process*(message), respectively, to initialise and deliver messages to it. The *IEE* communicates with the *Storage Service* through the **Storage** group of calls, namely *init*(), *put*(key, value), and *get*(key). These calls abstract the need, while describing our protocols, for the *IEE* to send a message to the *Server*, which in turn contacts the *Storage Service*, returning its output to the *IEE* at the end; and provide a clean key-value store-like interface.

***IEE* and *Client-IEE* Channel Initialisation Procedure** In the initialisation procedure, the structures needed for the protocols' operation are initialised. These structures include any steps required to initialise the *IEE* – **IEE**.*init*(), which is implementation dependent – and the *Client-IEE* channel. Both the *Server* and the *Storage Service* are assumed to be running and waiting for requests in this procedure, as their own initialisation is trivial.

The *Client* starts by generating a public-key encryption key pair (pk_c, sk_c), of which pk_c is hard-coded into the *IEE* program code, and will be used to bind the *IEE* with its *Client*. The program code is sent to the *Server*, which then runs the **IEE**.*init*() procedure with the received code; after that, the *Server* will only act as a proxy for all *Client-IEE* communications. The *Client* then starts the authentication procedure with the *IEE*, following the IEE algorithm of Barbosa et al. (2016), which provides the following properties: that an IEE is bound to a specific client, and can not answer to illegitimate clients; that a client must be able to attest it is communicating with a legitimate IEE, and the code being run in it is the one the client expects. To ensure these properties an authenticated key-exchange scheme is executed; at the end of which an authenticated symmetric key is shared between *Client* and *IEE*. The algorithm works as follows.

1. The *IEE* generates a public-key encryption key pair (pk_i, sk_i), sending pk_i to the *Client*, signed with an *IEE*-specific proof of attestation (also known as quote). This proof is implementation-dependent, and ensures the *Client* it was generated in a trusted IEE running the expected code.

2. The *Client* attests the received proof and generates a symmetric key k, encrypts it with pk_i and signs it with sk_c, sending it to the *IEE*.

3. The *IEE* decrypts and verifies the received key k, producing a confirmation message to the *Client*.

4. The *Client* receives and attests the confirmation message.

From here onwards, the *Client-IEE* channel is ready for secure point-to-point communications. The key-exchange algorithm provides two-way authentication and confidentiality guarantees. The use of an authenticated cipher in further communication can be used to provide integrity guarantees, and techniques like the use of nonces hinder attacks such as replaying.

3.4 BISEN: Boolean Isolated Searchable Encryption

In this section we describe BISEN (Boolean Isolated Searchable ENcryption), our scheme for secure Boolean querying over text data – *e.g.*, queries of the form $A \wedge (B \vee \neg C)$. BISEN consists of an index for retrieval of text documents, which can be queried by keywords, negations of keywords, or groups thereof, joined by conjunctions, disjunctions, and parentheses. Users can add, update, or remove words from text documents, as well as issue

queries to the system, receiving a set of matching document identifiers as response. This is known as the Boolean retrieval model (Manning et al., 2008, p. 4), which is a standard information retrieval model for large text databases search.

BISEN can support both exact-match and ranked responses. The latter has two main advantages: first, it allows for more relevant documents to be returned (*i.e.*, by ranking documents with more instances of the queried keyword higher); secondly, response length can be fixed, which hides the real response size (a common and severe leakage in SSE schemes (Bost and Fouque, 2017) leading to inference attacks), and improves communication latency for large databases.

In this section we will describe BISEN's protocols, first as an exact-match secure Boolean retrieval system (Section 3.4.1), provide their security analysis (Section 3.4.2), detail how the scheme can be extended for ranked responses (Section 3.4.3), and discuss design decisions and possible improvements (Section 3.4.4).

3.4.1 Protocols for Text Search

We will now describe the BISEN scheme, which consists of three protocols: *Setup* (which initialises data structures and communication), *Update* (which adds or deletes keywords in a given document[4]), and *Search* (which performs a Boolean query, resulting in a list of matching documents).

Algorithm 3.1 BISEN *Setup* protocol: Setup(λ)

Client:
1: $\mathsf{k}_F \leftarrow \mathsf{F}.gen(\lambda)$
2: $W \leftarrow [\,]$
3: msg $\leftarrow$ BISEN.*setup*(λ)
4: encMsg $\leftarrow$ SecureMsg.*enc*(msg)
5: Transport.*send*(Server, encMsg)

Server:
6: Transport.*recv*(encMsg)
7: **IEE**.*process*(encMsg)

IEE:
8: msg $\leftarrow$ SecureMsg.*dec*(encMsg)
9: $\lambda \leftarrow$ msg
10: $nrDocs \leftarrow 0$
11: $\mathsf{k}_E \leftarrow \Theta.gen(\lambda)$
12: **Storage**.*init*()

Setup BISEN's *Setup* protocol is described in Algorithm 3.1, taking the security parameter λ as input. The *Client* starts by initialising a PRF key k_F, which will be used for

[4]Adding and deleting keywords is seen as an update, the operation carrying an *op* code specifying the case.

keyword encryption, and a dictionary of counters W. These counters track the number of operations done over each keyword of the database, *i.e.* the number of times the word was added or deleted over all documents. A keyword's counter is used to generate all entries that need to be retrieved from the *Storage* when performing a search over that keyword[5].

The *Client* then generates the setup message for the *IEE* (line 3), sending it to the *Server* through the secure *Client-IEE* channel. The *Server* redirects the message to the *IEE*, which initialises a key k_E for authenticated symmetric encryption and *nrDocs*, the number of existing documents on the system – which is used to help resolve negation queries[6]. Finally, the *IEE* initialises the *Storage Service*, calling its respective initialisation procedure.

Algorithm 3.2 BISEN *Update* protocol: Update(op, w, id)

Client:
1: $\mathsf{k}_w \leftarrow \mathsf{F}.run(\mathsf{k}_F, w)$
2: $c \leftarrow \mathsf{W}[w]$
3: **if** $c = \perp$ **then**
4: $\quad c \leftarrow 0$
5: **else**
6: $\quad c \leftarrow c + 1$
7: **end if**
8: $\mathsf{W}[w] \leftarrow c$
9: msg $\leftarrow$ BISEN.*update*({op, id, c, k_w})
10: encMsg $\leftarrow$ SecureMsg.*enc*(msg)
11: Transport.*send*(Server, encMsg)

Server:
12: Transport.*recv*(encMsg)
13: **IEE**.*process*(encMsg)

IEE:
14: msg $\leftarrow$ SecureMsg.*dec*(encMsg)
15: {op, id, c, k_w} $\leftarrow$ msg
16: $l \leftarrow \mathsf{F}.run(\mathsf{k}_w, \mathsf{c})$
17: $\mathsf{id}^* \leftarrow \Theta.enc(\mathsf{k}_E, (l, \mathsf{op}, \mathsf{id}))$
18: **Storage**.*put*(l, id^*)
19: **if** id $>$ *nrDocs* **then**
20: $\quad$ *nrDocs* $\leftarrow$ *nrDocs* + 1
21: **end if**

Update The *Update* protocol for BISEN is described in Algorithm 3.2. An update takes as input the operation *op*, either *ADD* or *DEL*, the keyword W and document identifier

[5]For persistence, the *Client* may store W in disc storage.

[6]Negation resolution also requires that document identifiers have to be sequential. If a document is removed, it is still seen by the system as an empty document. This implies that if, during operation, all document's keywords were removed, it would still be returned in negated queries – which would be semantically correct.

id. We describe the protocol for a single keyword, extending it for multiple keywords is achieved by repeating the protocol for each keyword in batch, both at the *Client* (lines 1 to 8) and at the *IEE* (lines 15 to 21). Our delete operations are differentiated only by the operation code, and its effects are reflected only in the *Search* protocol, during *IEE* processing.

The *Client* starts by generating k_w, a pseudo-random transformation over the input keyword, which hides its size when sending it over the network. Then, the local counter value for the keyword is checked (line 2). If the keyword did not exist in the database before, the counter is initialised to zero, else it is incremented by one (line 6); the new value is then stored on the counter map W (line 8).

The *Client* sends a message containing the operation op, the document identifier id, the keyword transformation k_w and its counter c to the *IEE* through the *Client-IEE* channel. The *IEE* generates a label l over k_w and c with the PRF (line 16); by pairing the keyword transformation with its counter, the label is unique across the database, while also not revealing any information about its content to the untrusted components, and can be deterministically generated again when searching. A tuple $id*$ containing the label l, the operation op, and the document identifier id is encrypted with the authenticated symmetric cipher Θ (line 17); the label l is included for verification purposes when searching the database. These values are then inserted into the *Storage Service*, with l acting as key and $id*$ as value. Then, if the document identifier corresponded to a new document, the variable $nrDocs$ is incremented.

Search The *Search* protocol for BISEN is described in Algorithm 3.3. The protocol receives a query q as input, which corresponds to an arbitrary group of keywords, separated by conjunctions (AND operations) and disjunctions (OR operations); additionally, keywords can be negated (NOT operations) and be grouped by parentheses.

The *Client* starts by preprocessing q into a set of keywords $\bar{w}$ and the group of Boolean operators ϕ. Then, a dictionary of query counters C is initialised to hold the database counters for each keyword of the query; such counters are retrieved from W as in the *Update* protocol – by generating a k_w from the original word, and retrieving its counter from W. The dictionary C (of keywords and their frequency) and ϕ are then sent to the *IEE* via the secure channel.

The *IEE* starts by initialising Q, an auxiliary dictionary to help resolve the query, which will associate query keywords with their respective set of labels in the database. Resolving the query involves retrieving all database entries that refer to its keywords. The labels are generated in lines 16 to 23; for each keyword and respective counter a label is generated, similarly to the *Update* protocol, and the set of each keyword's labels is stored in Q (line 22). To request the labels to the *Storage Service*, a set L' is generated by grouping all labels in Q (line 24). Before performing the request, L' is randomly permuted (line 26), so that requests are not sequential, thus hindering correlations between *Storage* accesses.

Algorithm 3.3 BISEN *Search* protocol: Search(q)

Client:
1: $\{\bar{w}, \phi\} \leftarrow$ ProcessBooleanQuery(q)
2: $C \leftarrow [\ ]$
3: **for all** $w \in \bar{w}$ **do**
4: $\quad \mathsf{k}_w \leftarrow \mathsf{F}.run(\mathsf{k}_F, \mathsf{w})$
5: $\quad c \leftarrow W[w]$
6: $\quad C[\mathsf{k}_w] \leftarrow c$
7: **end for**
8: msg $\leftarrow$ BISEN.*search*($\{C, \phi\}$)
9: encMsg $\leftarrow$ SecureMsg.*enc*(msg)
10: Transport.*send*(Server, encMsg)

Server:
11: Transport.*recv*(encMsg)
12: **IEE**.*process*(encMsg)

IEE:
13: msg $\leftarrow$ SecureMsg.*dec*(encMsg)
14: $\{C, \phi\} \leftarrow$ msg
15: $Q \leftarrow [\ ]$
16: **for all** $\{\mathsf{k}_w, c\} \in C$ **do**
17: $\quad L \leftarrow \{\ \}$
18: $\quad$ **for all** $c_i \leftarrow 0 \ldots c$ **do**
19: $\quad\quad l \leftarrow \mathsf{F}.run(\mathsf{k}_w, c_i)$
20: $\quad\quad L \leftarrow L \cup \{l\}$
21: $\quad$ **end for**
22: $\quad Q[\mathsf{k}_w] \leftarrow L$
23: **end for**
24: $L' \leftarrow$ Flatten(Q)
25: $\Pi \leftarrow_{\$}$ RandomPermutation(λ)
26: $L' \leftarrow \Pi(L')$
27: $D' \leftarrow \{\ \}$
28: **for all** $l \in L'$ **do**
29: $\quad id* \leftarrow$ **Storage**.*get*(l)
30: $\quad D' \leftarrow D' \cup \{id*\}$
31: **end for**
32: $D \leftarrow \{\ \}$
33: **for all** $l' \in L'$; id* $\in D'$ **do**
34: $\quad (l, \mathsf{op}, \mathsf{id}) \leftarrow \Theta.dec(\mathsf{k}_E, \mathsf{id}^*)$
35: $\quad$ Verify(l,l')
36: $\quad D \leftarrow D \cup \{(\mathsf{op}, \mathsf{id})\}$
37: **end for**
38: $D \leftarrow \Pi^{-1}(D)$
39: $Q' \leftarrow$ Join(Q, D)
40: $Q' \leftarrow$ Filter(Q')
41: $R \leftarrow$ Resolve(ϕ, Q', $nrDocs$)
42: encAnswer $\leftarrow$ SecureMsg.*enc*(R)
43: Return *encAnswer* to Server.

Server:
44: Transport.*send*(Client, encAnswer)

Client:
45: Transport.*recv*(encAns)
46: $R \leftarrow$ SecureMsg.*dec*(encAns)

Labels in L' are requested to the *Storage Service*, and obtained values (set of all $id*$) are stored in D'.

All values in D' are iterated in lines 33 to 37. A set D will hold pairs ($operation$, $documentid$) after decryption. In this iteration, obtained values are first decrypted to obtain the tuple containing the label verification l, the operation code op, and the document identifier id. Verification of the obtained value is performed by comparing the requested label l' to the label in the tuple l (line 35); if the labels do not match, the *IEE* can repudiate the answer and terminate, as the *Storage* provided a tampered result. Otherwise, op and id are stored as a pair in D. D is then permuted to the original request order, so that its contents can be joined with Q, to form a dictionary Q' (lines 38 and 39), which associates query keywords to the set of pairs ($operation$,$documentid$) where such keyword occurs. The Filter algorithm (line 40) takes into account the op code and the order of counters to decide whether a keyword exists in a document or not: for a given document, the keyword's state – existing or deleted – is determined by the highest, *i.e.* last, counter value

referring to that document; Filter deletes outdated pairs from Q'.

Query resolution can then be performed simply as a group of set operations (line 41), since each keyword is associated with the set of documents where it occurs. Boolean processing is performed by first translating the query to Reverse Polish Notation and applying the Shunting Yard algorithm (Dijkstra, 1961). Negation is performed by searching for the non-negated version of the keyword and then inverting the resulting set (we achieve this by generating an auxiliary set representing all documents from 0 to $nrDocs$).

Finally, the resulting set of documents, R, is returned to the *Client* via the secure channel (lines 42 to 46).

3.4.2 Security Analysis

In this section we will informally analyse the security of the BISEN scheme. We consider both the *Client* and *IEE* to be secure by definition, and as such providing full confidentiality, integrity and authentication guarantees. We will start by analysing each untrusted component and the data it has access to, then both *Client-IEE* and *IEE-Storage* channels, and finally analyse leakage and privacy guarantees given by the scheme.

Server The *Server* is responsible for acting as proxy for the *Client-IEE* and *IEE-Storage* channels. Since data travelling through such channels is secured, the only possible attack is that of a denial-of-service. The *Server* only has access to encrypted data, and cannot perform tampering, as all data is authenticated and verified both at the *IEE* and *Client*.

Storage Service The *Storage Service* consists of a key-value store with regular *put* and *get* operations. The *IEE* inserts encrypted $(label, value)$ pairs into the *Storage*; the attacker can choose to not insert such pair, to insert a *label* with a non-matching value, to tamper either the *label* or the $value$, or to insert additional pairs. When retrieving data (*Search* operation), the *IEE* will generate labels based on the counter value for each searched keyword. In the case a pair is missing, the *IEE* detects one of the requested labels was not retrieved and halts operation; when decrypting retrieved pairs, an authenticated cipher is used, so that the $value$ can be checked for tampering; if the $value$ is untampered, the copy of the *label* in it is checked and, if it does not match the requested *label*, such *label* is associated with an erroneous $value$ and operation is also halted. Finally, extra pairs do not affect the system, as they would never be requested by the *IEE*. Different pairs on the *Storage* can not be related with each other, since *labels* are composed of a unique $(word, id)$ pair encrypted with a PRF, thus making any two entries noncorrelatable; the respective document identifier in the $value$ is unique and encrypted with a symmetric cipher, thus also being completely independent of other values mentioning the same document.

***Client-IEE* Channel** The channel is established with the *Server* acting as a proxy for data communication. The *IEE* is hard-coded with the *Client*'s public key, ensuring that

the *IEE* will always communicate with the same *Client*, and thus the attacker cannot suddenly interact with the *IEE*, forcing it to answer malicious operations. To leverage IEE guarantees, the *Client* must be sure that it is communicating with a legitimate IEE, and that the IEE is running the expected program, of which the *Client* has a digest. The *Client* does so by verifying that the attestation proof (quote) received during the initialisation procedure is signed with an IEE-specific key (in practice provided by its vendor), and that it also contains the correct digest of the program expected to be running. These verifications are performed during the key-exchange algorithm (Section 3.3), and thus the *Client* can be sure of confidentiality, integrity and authentication guarantees at the start of communications. Given that both endpoints of the channel are considered secure, it follows that the channel remains secure throughout system operation.

Leakage of Protocols We define our leakage in function of data sent through and to untrusted components. The protocol being executed is always leaked, as the type of access to the *Storage Service* reveals it.

In the *Update* protocol the *Client* sends a fixed-width message to the *IEE*, as the only variable-length component – the word – is encrypted with a PRF before leaving the *Client*. Therefore, a single *Update* operation also only produces an entry in the *Storage Service*; while a batch of *Update* operations reveals the number of updated keywords, but not their individual lengths. Moreover, additions and deletes are indistinguishable, as they are only defined by a single operation code, which is sent encrypted with the rest of the message; processing of operations is only done *IEE*-side when performing a *Search*.

When performing a *Search* operation, the *Client* sends an encrypted set of keywords and respective counters, together with the Boolean form of the query. As with the *Update* protocol, we leak the number of keywords in the input of such operation. Most of the query processing is done inside the *IEE*; the *IEE* requests the needed pairs from the *Storage Service* in a random order, so as to hide correlations between same instances of the keyword in different documents, for example. *Storage Service* operations produce *access leakage*, as the memory positions accessed are revealed; avoiding such leakage would involve ORAM techniques. When answering to the *Client*, the *IEE* reveals the size of the answer, *i.e.* the number of documents included in the response.

In comparison with the state-of-the-art (Table 2.1 from Section 2.3.4), BISEN only produces *access leakage*, as an adversary cannot distinguish whether two searches that produced the same *access pattern* are a repetition of the same search, or two different searches that referred to the same keywords (*e.g.*, a conjunctive and a disjunctive query with the same operands).

Backward and Forward Privacy BISEN preserves both backward and forward privacy. Backward privacy is based on the definition of Bost et al. (2017), and ensures that adding

a keyword w, and subsequently deleting it, is indistinguishable when performing successive searches over w. BISEN's *Storage* is monotonically crescent, with deletions corresponding to a new entry in the *Storage*; therefore, when searching for w, the number of retrieved pairs will never be smaller than with the previous search, thus hiding whether the keyword was added to new documents or deleted from previous ones.

To preserve forward privacy, BISEN must ensure that previous searches can not be related with new updates (Bost, 2016), *i.e.*, that an attacker can not know whether a newly updated keyword was in a previous search. Since all entries of the *Storage* are independent from one-another (are encrypted with a PRF), and all cryptographic computations are done either in the *Client* or the *IEE*, an attacker does not learn if a new pair would be retrieved by repeating an old query, and has no access to tokens or cryptographic keys capable of doing so. The *Storage Service* only observes accesses to the index.

3.4.3 Extending BISEN for Ranked Retrieval

In this section we describe how to extend the presented scheme (Section 3.4.1) to allow for ranked retrieval. The exact-match version's *Search* protocol returns a set of documents matching the given query; if it is too broad and the database is large, the resulting set is substantial, possibly incurring in high network overhead. Moreover, exact-match retrieval can also leak the type of operation for two queries with the same keywords: *e.g.*, with queries $A \wedge B$ versus $A \vee B$, the first query will probably result in a smaller document set than the second one. By fixing a response size, the *IEE* can pad or truncate the result set, always returning messages of equal size to the *Client*[7]. Moreover, by introducing ranks search results become more relevant to the user.

To allow for document ranking, metrics like *TF–IDF* can be employed. This scoring function requires keyword frequency to be known for *TF*; the total number of documents and the counter of each query keyword for *IDF*. Of these, the only information not available in the exact-match version is keyword frequency, the addition of which we detail now.

Changes to the *Update* Protocol In the *Update* protocol (first described in Algorithm 3.2), we change the input op, a value indicating whether the keyword was added or deleted, to a frequency value f, indicating the frequency of the keyword in the document id. An op of type ADD becomes an f of a given positive frequency, while a DEL is equivalent to $f = 0$. As with the exact-match version, when referring to a keyword w in document id, the prevailing frequency will be that of the highest counter referring to the document.

Changes to the *Search* Protocol The *Search* protocol takes a new argument ρ (a threshold to normalise the response size), remaining otherwise similar to the original version

[7] Accesses to the *Storage Service* are already indistinguishable for both queries, in either exact-match or ranked version.

Algorithm 3.4 BISEN *Search* document scoring.

```
1:  IDF ← CalcIDF(C, nrDocs, Q')
2:  S ← [ ]
3:  for all k_w ∈ Q' do
4:      k_{w_idf} ← IDF[k_w]
5:      for all (f, id) ∈ Q'[k_w] do
6:          if id ∈ R then
7:              s ← S[id]
8:              if s = ⊥ then
9:                  s ← 0
10:             else
11:                 s ← s + f * k_{w_idf}
12:             end if
13:             S[id] ← s
14:         end if
15:     end for
16: end for
17: S' ← SortByValueDesc(S)
18: S' ← PadOrTruncate(S', ρ)
```

(first described in Algorithm 3.3) up to line 41. All occurrences of op are also changed to f.

The scoring procedure in the *IEE* is inlined between lines 41 and 42 of Algorithm 3.3, when R contains the response documents and before the response is sent to the *Client*.

The procedure is presented in Algorithm 3.4. It starts by calculating the *Inverse Document Frequency* of each keyword in the query, using C (to list such query keywords), the total number of documents in the database ($nrDocs$), and the global frequency of each keyword – deduced from Q', since, after Filter, a keyword k_w will have global frequency equal to the cardinality of the set in $Q'[w]$. IDF is calculated via $IDF(k_w) = \log \frac{nrDocs}{|Q'[w]|}$, as described in Section 2.2.3.

Then, all keywords of the query are iterated (line 3) and, for each occurrence (f, id) of the keyword (line 5), the score of document id is updated with *TF–IDF* (line 11), provided id belongs to the response set R (line 6).

A dictionary S', associating documents to their scores, is generated by ordering S in descending order of score and padding or truncating results to ρ (line 18), hiding the real response size. The resulting S' is then returned to the *Client* instead of R.

A Caveat with *nrDocs* and Rank Scoring Designing negation queries requires the knowledge of the full document set, a purpose *nrDocs* serves. Our protocol interface does not allows us to know when a document was fully deleted, as only individual keywords can be deleted, and full knowledge of every document's keywords is not a *Client* or

IEE requirement[8]. However, by using *nrDocs* for *Inverse Document Frequency*, we cannot be sure whether its value is stale or not, as all keywords in one or several documents might have been deleted. This implies a possible deviation of the true score of a document; however, since all scoring will be done using the same *nrDocs* value, relative rank values will still be correct in regard to their relations between each other.

Security Analysis Extending BISEN for ranked search only implies data structures changes at the *Client* and the *IEE*; therefore, security guarantees given previously (Section 3.4.2) hold equally. Replacing operation codes for keyword frequency in data stored at the *Storage Service* might imply changing the length in bits of $\mathsf{id}*$; however, the value is encrypted, and the add/update/remove behaviour remains the same as previously; as such holding the same security guarantees as the exact-match version. Furthermore, while in exact-match BISEN the response length to the *Client* could help an attacker infer the operator types, with the fixed-width responses of the ranked version such information can not be inferred anymore.

When performing searches the type of Boolean operator is no longer leaked, as responses become fixed-width.

3.4.4 Discussion

One of the main design principles in BISEN is to reduce storage needs in the *IEE* module, since some implementations limit its memory and disc capabilities; usage of the *Storage Service* allows us to delegate such task to an insecure module keeping strong security guarantees and minimal leakage. Notwithstanding, during a query's processing the *IEE* still requests and temporarily stores all relevant pairs for that query; in very large databases, the *IEE* might not be able to contain the full data structures in it. Queries would benefit from incremental processing, where the *IEE* would retrieve part of the relevant data each iteration and resolve part of the query immediately, discarding used entries meanwhile; and merging results with the previous iterations' work.

Since no true deletions occur in BISEN, its *Storage Service* data is always increasing, which can be a hindrance on update-intensive systems, as the IEE would be forced to retrieve, and temporarily store, all previous updates up to the point of querying. While deleting pairs from the *Storage* would break privacy assumptions, by re-indexing the database all outdated entries could be removed, and the use of fresh keys would make correlations with the old index impossible. This operation, however, would be computationally expensive, and involve both the *Client* and *IEE*'s processing.

The dictionary of counters W in the *Client* could be migrated to the *IEE* in the scheme, as all operations done with counters are *IEE*-side. Being stored in the *Client* is a more scalable approach if the database contains a large variety of words, and memory space for

[8] Unless the *Client* knows the full document and is sure every of its keywords was deleted – if so, it can inform the *IEE* and the caveat is non-existent.

query resolution is a concern; for medium and small databases such concern might not be significant. Having W on the *IEE* can also facilitate a multi-user solution, as counters would need to be synchronised across clients in the current version. Conversely, having W on the *Storage Service* would not retain the same security guarantees as the previous options. Even if encrypted (with schemes such as homomorphic encryption), W only has two operations, increment by one and get; therefore, updates over the same keyword would be revealed while updating the counter, and relations between inserted pairs into the *Storage* could be inferred by increments to the same position of W.

For persistence purposes, the key k_E, generated and only known to the *IEE*, could be shared with the *Client*. In case of a denial-of-service attack over the *IEE*, and assuming the *Storage* is kept intact and still available, the *Client* could instantiate a new *IEE* and transmit its k_E, enabling the use of the same *Storage Service* and its entries.

Regarding BISEN's ranked version: we consider an absolute frequency f to be stored within a keyword's entry in the *Storage*. This approach requires the *Client* to know the frequency of a keyword for an update operation. Having incremental frequency values on *Storage* would rid the *Client* of this requirement; updates would contain the change in frequency rather than its current value[9]. The Filter operation would keep track of the current frequency until the last keyword's counter is reached.

Other metrics could be used to score documents in the ranked version, such as *BM25* (Manning et al., 2008, Section 11.4.3) or the Vector Space Model (Salton et al., 1975). Some metrics depend on the typology of documents in the database and, as such, the protocols may be changed to support a different scoring function – if no further information is needed from the database, scoring functions can be used interchangeably and independently of the rest of the scheme.

3.5 VISEN: Visual Isolated Searchable Encryption

We will now describe VISEN (Visual Isolated Searchable ENcryption), our scheme for secure Content-Based Image Retrieval (CBIR). VISEN consists of an index for image documents: users can add or remove images to the database, and query by using example images, with the response being the most similar images to that query. We consider image files to be immutable (only additions and deletions are supported) throughout the system's operation, but in end of the section (Section 3.5.3) we discuss an extension to a scheme supporting updates.

VISEN works by adopting a Bag of Visual Words (BoVW) method (Nistér and Stewénius, 2006). Images are described by sets of feature vectors, which are extracted via an appropriate algorithm – like SURF (Bay et al., 2008) or SIFT (Lowe, 2004). These feature vectors represent interest points in the image, defined by sudden changes in colour, intensity, or texture. Since each image amounts to large sets of feature vectors, and each vector

[9]Performing a delete would not be an update of $f = 0$, but could be done by inserting a special value, such as $-\infty$. A further positive update over that keyword would signify its restoration to the document

is highly-dimensional (usually between 64 and 128 dimensions), linear searching over a database of such vectors becomes impractical. Therefore, to make CBIR schemes practical, clustering algorithms can be used to group a large vector space into small sets, known as Bags of Visual Words, grouping similar vectors together. Clustering can be achieved through different techniques, such as machine learning training algorithms, like k-means (MacQueen, 1967), binarisation algorithms (Liu et al., 2014b), or Locality-Sensitive Hashing (LSH) (Loi et al., 2013); these algorithms output a codebook of clusters (each cluster being a vector of the same form of feature vectors); inserted vectors are then fitted to the most similar cluster of that codebook.

Some of the codebook generation techniques, like k-means, may require a training phase, where example vectors are used to generate clusters; other techniques, such as LSH, do not need such phase. In our experimental evaluation (Section 5.3.1), we discuss the precision of different techniques (two variants of k-means and LSH), and their respective performance. Regardless of the used technique, adding images to VISEN implies the feature vector extraction step, followed by the clustering of those vectors. A frequency histogram of the image's clusters is produced; for each entry, the frequency will indicate the number of original image vectors that were put into that cluster. Searching for images implies generating a similar histogram, comparing it with the existing database, and returning the most similar images ranked by metrics like *TF–IDF*.

If the codebook generation technique requires training, such step is done before the system's normal operation. This step may be computationally expensive, but it is executed only once, or if the dataset changes significantly. In our scheme, we consider three different codebook generation approaches: *Traditional K-means*, where all vectors can be iterated several times over, *Online K-means*, where each vector is only iterated once, and LSH, where vectors are hashed into a cluster without previous training.

Bags to which features are clustered into are akin to keywords in BISEN, *i.e.* each feature is a keyword, and images are a set of keywords with varying frequencies, similarly to ranked BISEN. VISEN adapts BISEN's techniques, making some specific design choices to adapt to the image domain (*e.g.*, the keyword space of VISEN is bound by the chosen number of clusters during the scheme's initialisation).

In the remainder of the section we will present the secure CBIR scheme (Section 3.5.1), provide its security analysis (Section 3.5.2), and discuss possible improvements and future work (Section 3.5.3).

3.5.1 Protocol for Content-Based Image Retrieval

The VISEN scheme consists of a *Setup* protocol (which initialises the needed data structures and communication), followed by a *codebook generation phase* (to create a codebook for feature clustering) and an *operating phase* (where images can be added, removed, and queries performed). In the *codebook generation phase* the used protocols depend on the chosen algorithm, while in the *operating phase* the protocols (*Add*, *Remove*, and *Search*) are

equal regardless of the employed codebook generation algorithm.

Before describing such protocols, we present the three different versions for the *codebook generation phase*:

- **Traditional K-means** This version of k-means (Lloyd, 1982) requires a global vision of the training dataset, and thus low-latency large memory storage. Therefore, we execute the algorithm at the *Client*, sending the resulting codebook to the *IEE*. We denote this protocol as *TrainKmeans*.

- **Online K-means** Proposed by MacQueen (1967); training data is sent to the *IEE* as it is incrementally extracted from the training dataset, and the codebook is updated at the *IEE* immediately. This algorithm does not require a global vision of the dataset, and can therefore be executed in our resource constrained IEE. We denote this protocol as *TrainOnlineKmeans*.

- **Locality-Sensitive Hashing** Based on Loi et al. (2013); clusters are generated semi-randomly in the *IEE*; no training dataset is needed. This protocol is denoted *GenerateClustersLSH*.

After executing either version, cluster centroids are stored in the *IEE*, and are used for the *operating phase* algorithms.

Algorithm 3.5 VISEN *Setup* protocol: Setup(λ, k, $threshold$)

Client:
1: $nrClusters \leftarrow k$
2: $\omega \leftarrow threshold$
3: msg $\leftarrow$ VISEN.*setup*(λ, k)
4: encMsg $\leftarrow$ SecureMsg.*enc*(msg)
5: Transport.*send*(Server, encMsg)

Server:
6: Transport.*recv*(encMsg)
7: **IEE**.*process*(encMsg)

IEE:
8: msg $\leftarrow$ SecureMsg.*dec*(encMsg)
9: $\{\lambda, k\} \leftarrow$ msg
10: $nrClusters \leftarrow k$
11: $\mathsf{k}_F \leftarrow \mathsf{F}.gen(\lambda)$
12: $\mathsf{k}_E \leftarrow \Theta.gen(\lambda)$
13: $C \leftarrow [\,]$
14: $\mathsf{W} \leftarrow [\,]$
15: $nrImgs \leftarrow 0$
16: **Storage**.*init*()

Setup The *Setup* protocol for VISEN is described in Algorithm 3.5. The protocol receives the security parameter λ, the number of clusters k, and an image descriptor threshold *threshold* as input, of which the latter two are stored by the *Client*, respectively, as $nrClusters$ and ω; the number of clusters is used for *Client*-side training algorithms; the descriptor threshold is used by the *Client* to control the number of feature vectors an image generates, or the sensitivity of the descriptor algorithm, and its value depends on the implementation used. The *Client* then generates a setup message containing the number of clusters for the *IEE* (line 3), sending it to the *Server*, and from there onwards to the *IEE*, through the secure *Client-IEE* channel.

As with BISEN, the *Server* redirects the message to the *IEE* for processing (line 7). The *IEE* stores the number of clusters and generates a PRF key k_F (line 11), which will be used to generate keywords based on each cluster identifier, and a key k_E for authenticated symmetric encryption (line 12). The *IEE* also initialises a codebook C and a dictionary of counters W[10].

Counters from W are used to generate all entries that need to be retrieved from the *Storage* when performing a search that includes such cluster. Conversely to BISEN, counters are stored in the *IEE*, as clusters, *i.e.* keywords, are not known by the *Client*, since the approximation procedure is done *IEE*-side, following our principle of executing heavier computations on Cloud resources. Finally, the *IEE* initialises $nrImgs$, which counts the number of images in the database[11], and the *Storage Service*, calling its respective initialisation procedure.

3.5.1.1 Codebook Generation Phase Procedures

TrainKmeans Codebook generation for the *Traditional K-means* approach is presented in Algorithm 3.6. It receives a set of training images $imgs$ as input; for each image, the *Client* extracts its feature vectors (line 3) into a set D. These images can come from a separate training dataset or from a subset of the images to be added to the database in the *operating phase*. Training is performed via the standard k-means algorithm (line 6) over the feature vector dataset, resulting in a codebook $clusters$. These are sent to the *IEE*, which stores them in C (line 14).

TrainOnlineKmeans The *Online K-means* approach is described in Algorithm 3.7. This protocol takes a single image img as input, whose feature vectors are extracted by the *Client* and sent to the *IEE*. The *IEE* receives the set of feature vectors and performs the *Online K-means* operation (line 9): if the codebook is empty, it is initialised to the received vectors; else, already-existing centroids are adjusted with the newly-received vectors. After visiting each feature vector once, the *IEE* can discard it. This protocol is executed several times, once for each image in the training dataset.

[10] Clusters are identified numerically, from 0 to k. C associates a cluster to its feature vector, *i.e.* centroid, and W tracks the overall frequency of each cluster in the database.

[11] This variable is used for the scoring function of the *Search* protocol.

Algorithm 3.6 VISEN codebook generation protocol for k-means: TrainKmeans($imgs$)

Client:
1: $D \leftarrow \{\}$
2: **for all** $i \in imgs$ **do**
3: $F \leftarrow$ ExtractFeatures(i, ω)
4: $D \leftarrow D \cup F$
5: **end for**
6: $clusters \leftarrow$ Kmeans($nrClusters$, D)
7: msg $\leftarrow$ VISEN.*train_kmeans*($clusters$)
8: encMsg $\leftarrow$ SecureMsg.*enc*(msg)
9: Transport.*send*(Server, encMsg)

Server:
10: Transport.*recv*(encMsg)
11: **IEE**.*process*(encMsg)

IEE:
12: msg $\leftarrow$ SecureMsg.*dec*(encMsg)
13: $clusters \leftarrow$ msg
14: $C \leftarrow clusters$

Algorithm 3.7 VISEN codebook generation protocol for online k-means: TrainOnlineKmeans(img)

Client:
1: $F \leftarrow$ ExtractFeatures(img, ω)
2: msg $\leftarrow$ VISEN.*train_onlinekmeans*(F)
3: encMsg $\leftarrow$ SecureMsg.*enc*(msg)
4: Transport.*send*(Server, encMsg)

Server:
5: Transport.*recv*(encMsg)
6: **IEE**.*process*(encMsg)

IEE:
7: msg $\leftarrow$ SecureMsg.*dec*(encMsg)
8: $F \leftarrow$ msg
9: $C \leftarrow$ OnlineKmeans($nrClusters$, C, F)

Algorithm 3.8 VISEN codebook generation protocol for LSH: GenerateClustersLSH()

Client:
1: msg ← VISEN.*train_lsh*()
2: encMsg ← SecureMsg.*enc*(msg)
3: Transport.*send*(Server, encMsg)

Server:
4: Transport.*recv*(encMsg)
5: **IEE**.*process*(encMsg)

IEE:
6: msg ← SecureMsg.*dec*(encMsg)
7: C ← GenerateStdVectors($nrClusters$)

GenerateClustersLSH The LSH approach for the *codebook generation phase* is presented in Algorithm 3.8. The protocol takes no input parameters, and consists of randomly generating cluster centroids on the *IEE* (line 7); the values for these centroids following a standard distribution.

3.5.1.2 Operating Phase Procedures

In this section we will describe the three *operating phase* protocols: *Add*, *Remove*, and *Search*. Beforehand, we will describe an alteration to our **Storage** primitive, needed to ensure *backward privacy* in the setting of VISEN.

VISEN considers images as indivisible elements, with each image addition or removal producing a varying number of entries (*i.e.*, keywords) in the *Storage Service*, while BISEN considered single-keyword operations. *Add* operations produce a number of entries dependant on the number of different clusters the image's features produces; *Remove* operations consist of resetting all clusters to zero (thus inserting a fixed number of entries in *Storage*), as a removal does not take the original image as input.

Preserving *backward privacy* implies making additions and removals indistinguishable; as such, we consider an *entry buffer* in the *IEE*: entries that are to be written to the *Storage Service* are held in this buffer, up to a given limit based on available memory; entries are then written in batch to the *Storage Service* when the limit is reached. This forces *Storage* writes to always be of a fixed length. Read operations (when searching) are made by first consulting the *entry buffer*, the *Storage Service* being contacted only if the requested entry is not present in the IEE *entry buffer*.

In the description of our protocols, our *entry buffer* is abstracted by the **Storage** primitives, which contact the *buffer* before the *Storage Service*.

Add The *Add* protocol for VISEN is presented in Algorithm 3.9, and takes an image *img* and its identifier id as input. The *Client* performs a feature extraction step (line 1) and sends the resulting set of vectors F and the image identifier to the *IEE*. The cardinality of

Algorithm 3.9 VISEN *Add* protocol: Add(img, id)

Client:
1: $F \leftarrow$ ExtractFeatures(img, ω)
2: msg $\leftarrow$ VISEN.*add*({id, F})
3: encMsg $\leftarrow$ SecureMsg.*enc*(msg)
4: Transport.*send*(Server, encMsg)

Server:
5: Transport.*recv*(encMsg)
6: **IEE**.*process*(encMsg)

IEE:
7: msg $\leftarrow$ SecureMsg.*dec*(encMsg)
8: {id, F} $\leftarrow$ msg
9: $H \leftarrow$ ApproxDescriptors(F)
10: $P \leftarrow \{\}$
11: **for all** $i \leftarrow 0 \ldots nrClusters$ **do**
12: $f \leftarrow H[i]$
13: **if** $f > 0$ **then**
14: $c \leftarrow \mathsf{W}[i]$
15: **if** $c = \perp$ **then**
16: $c \leftarrow 0$
17: **else**
18: $c \leftarrow c + 1$
19: **end if**
20: $\mathsf{W}[i] \leftarrow c$
21: $\mathsf{k}_c \leftarrow \mathsf{F}.run(\mathsf{k}_F, i)$
22: $l \leftarrow \mathsf{F}.run(\mathsf{k}_c, c)$
23: $\mathsf{id}^* \leftarrow \Theta.enc(\mathsf{k}_E, (l, f, \mathsf{id}))$
24: $P \leftarrow P \cup \{(l, id*)\}$
25: **end if**
26: **end for**
27: $\Pi \leftarrow_{\$}$ RandomPermutation(λ)
28: $P' \leftarrow \Pi(P)$
29: **for all** $(l, id*) \in P'$ **do**
30: **Storage**.*put*(l, id^*)
31: **end for**
32: $nrImgs \leftarrow nrImgs + 1$

F depends on ω, which represents an upper bound for the number of features produced; to reduce leakage, F is padded to its upper bound to produce fixed-width adds. The *IEE* approximates each vector from F to its closest cluster (line 9), using a metric like Euclidean distance; this operation yields a histogram H that associates each cluster with its frequency in the image img. All data that is to be sent to the *Storage* will be stored in a set P (line 10). The *IEE* then iterates over all clusters whose frequency is non-null (line 13); for each one the *IEE* retrieves its counter value, initialising or incrementing it as needed (lines 14 to 20). Similarly to BISEN, we create a unique (across the database) label l as a pseudo-random transformation over the cluster identifier and the current counter value (lines 21 and 22); the label is designed so as to be easily re-generated by the *IEE* when searching. A tuple containing l (for verification purposes), the image's cluster frequency and its document identifier is encrypted with the authenticated symmetric cipher Θ, yielding $id*$ (line 23); the pair $(l, id*)$ is then stored in P.

To avoid inference over the entries' ordering, a random permutation is performed over P (line 28) yielding P'. All pairs $(l, id*)$ in P' are iterated and added to the *entry buffer*, and eventually to the *Storage Service*. The image counter global variable $nrImgs$, used for scoring, is then incremented (line 32).

Algorithm 3.10 VISEN *Remove* protocol: Remove(id)

Client:
1: msg ← VISEN.*remove*(id)
2: encMsg ← SecureMsg.*enc*(msg)
3: Transport.*send*(Server, encMsg)

Server:
4: Transport.*recv*(encMsg)
5: **IEE**.*process*(encMsg)

IEE:
6: msg ← SecureMsg.*dec*(encMsg)
7: {id, F} ← msg
8: $P \leftarrow \{\}$
9: **for all** $i \leftarrow 0 \ldots nrClusters$ **do**
10: $c \leftarrow \mathsf{W}[i]$
11: **if** $c = \perp$ **then**
12: $c \leftarrow 0$
13: **else**
14: $c \leftarrow c + 1$
15: **end if**
16: $\mathsf{W}[i] \leftarrow c$
17: $\mathsf{k}_c \leftarrow \mathsf{F}.run(\mathsf{k}_F, i)$
18: $l \leftarrow \mathsf{F}.run(\mathsf{k}_c, c)$
19: $\mathsf{id}^* \leftarrow \Theta.enc(\mathsf{k}_E, (l, 0, \mathsf{id}))$
20: $P \leftarrow P \cup \{(l, id*)\}$
21: **end for**
22: $\Pi \leftarrow_{\$} \mathsf{RandomPermutation}(\lambda)$
23: $P' \leftarrow \Pi(P)$
24: **for all** $(l, id*) \in P'$ **do**
25: **Storage**.*put*(l, id^*)
26: **end for**
27: $nrImgs \leftarrow nrImgs - 1$

Remove The *Remove* protocol for VISEN is presented in Algorithm 3.10, and takes an image identifier id as input. Our intuition for the protocol is as follows: the *Client* is not required to keep an image stored locally[12]; the *Client* also does not know an image's clusters, since these are calculated at the *IEE* – hence, deleting can not be done by sending all of that image's clusters to the *IEE*, so that their frequency can be set as null for that image; an approach similar to BISEN is also not feasible in the image domain, as users are not expected to delete individual features directly.

Nevertheless, taking into account that the number of clusters, *i.e.* keywords, is fixed and small (compared, for example, with the number of possible words in BISEN), removing an image can be done by instead resetting all clusters to a null frequency – even if they were not in the image to start with.

As such, the *Client* starts by sending to the *IEE* the id of the image to be deleted. In the *IEE*, tuples $(l, id*)$ with $f = 0$ (line 19) are generated for each possible cluster (and each cluster's counter incremented), and added to the *entry buffer*, and eventually to the *Storage Service*. The most recent frequency value for each cluster of that image will be 0, and thus the image is effectively deleted, as searches will ignore null frequencies.

Search The *Search* protocol for VISEN is presented in Algorithm 3.11, taking an image img and a response size threshold ρ as input. The *Client* starts by extracting the image's

[12] In an alternative approach, the *Client* could retrieve the image from an external storage (where it would be encrypted with standard algorithms) and send it to the *IEE*, where the approximation step would be performed to infer which clusters would need to be removed.

Algorithm 3.11 VISEN *Search* protocol: Search(img, ρ)

Client:
$F \leftarrow$ ExtractFeatures(img, ω)
msg $\leftarrow$ VISEN.*search*($\{F, \rho\}$)
encMsg $\leftarrow$ SecureMsg.*enc*(msg)
Transport.*send*(Server, encMsg)

Server:
Transport.*recv*(encMsg)
IEE.*process*(encMsg)

IEE:
msg $\leftarrow$ SecureMsg.*dec*(encMsg)
$\{F, \rho\} \leftarrow$ msg
$H \leftarrow$ ApproxDescriptors(F)
$Q \leftarrow [\,]$
$I \leftarrow \{\,\}$
for all $i \leftarrow 0 \ldots nrClusters$ **do**
  if $H[i] > 0$ **then**
    $L \leftarrow \{\,\}$
    $\mathsf{k}_c \leftarrow \mathsf{F}.run(\mathsf{k}_F, i)$
    $c \leftarrow \mathsf{W}[i]$
    for all $c_i \leftarrow 0 \ldots c$ **do**
      $l \leftarrow \mathsf{F}.run(\mathsf{k}_c, c_i)$
      $L \leftarrow L \cup \{l\}$
    end for
    $Q[\mathsf{k}_c] \leftarrow L$
    $I \leftarrow I \cup \{i\}$
  end if
end for
$L' \leftarrow$ Flatten(Q)
$\Pi \leftarrow_{\$}$ RandomPermutation(λ)
$L' \leftarrow \Pi(L')$
$D' \leftarrow \{\,\}$
for all $l \in L'$ **do**
  $id* \leftarrow$ **Storage**.*get*(l)
  $D' \leftarrow D' \cup \{id*\}$
end for
$D \leftarrow \{\,\}$
for all $l' \in L'$; id* $\in D'$ **do**
  $(l, f, \mathsf{id}) \leftarrow \Theta.dec(\mathsf{k}_E, \mathsf{id}^*)$
  Verify(l,l')
  $D \leftarrow D \cup \{(f, \mathsf{id})\}$
end for
$D \leftarrow \Pi^{-1}(D)$
$Q' \leftarrow$ Join(Q, D)
$Q' \leftarrow$ Filter(Q')
$IDF \leftarrow$ CalcIDF($nrClusters$, $nrImgs$, Q')
$S \leftarrow [\,]$
for all $k \in Q'$; $i \in I$ **do**
  $c_{idf} \leftarrow IDF[i]$
  $c_{q_f} \leftarrow H[i]$
  for all $(f, \mathsf{id}) \in k$ **do**
    $s \leftarrow S[\mathsf{id}]$
    if $s = \perp$ **then**
      $s \leftarrow 0$
    else
      $s \leftarrow s + c_{q_f} * f * c_{idf}$
    end if
    $S[\mathsf{id}] \leftarrow s$
  end for
end for
$S' \leftarrow$ SortByValueDesc(S)
$S' \leftarrow$ PadOrTruncate(S', ρ)
encAnswer $\leftarrow$ SecureMsg.*enc*(S')
Return *encAnswer* to Server.

Server:
Transport.*send*(Client, encAnswer)

Client:
Transport.*recv*(encAns)
$S' \leftarrow$ SecureMsg.*dec*(encAns)

feature vectors into F and sending the latter to the *IEE*, together with threshold ρ. The *IEE* performs the same cluster approximation step as in the *Add* protocol, and initialises two auxiliary data structures, Q and I. Q is a dictionary associating clusters with their respective set of labels; I is a set holding all cluster identifiers whose frequency in the image is non-null – used when scoring results.

On line 12, the *IEE* starts by iterating over all non-null clusters, to generate all labels to be requested from the *Storage Service*. As with the *Add* protocol, each is a pseudo-random transformation over the cluster identifier and its current counter value. Labels are then grouped as a set L' (line 25), and are randomly permuted (line 27), so that possible correlations between labels of the same cluster are hidden. The request to the *Storage Service* is performed (lines 29 to 32), with the values corresponding to each label being stored in set D'. These values are decrypted in lines 34–38; each value is checked for integrity and authenticity by the authenticated cipher (line 35), and the Verify algorithm (line 36) ensures the retrieved value belongs to the requested label; the *IEE* can terminate processing due to tampering otherwise.

The obtained values are sorted to the original order in L' (line 39), and then joined to Q (line 40), yielding a dictionary Q' associating clusters to their set of pairs (f, id), *i.e.* images containing such cluster. The Filter operation (line 41) works as with BISEN – it removes values from Q' that have been superseded by more recent updates.

Having the query results, the protocol then performs their scoring. To use the *TF–IDF* metric, the *IEE* first calculates the *Inverse Document Frequency*, based on the global frequency of each cluster (line 42); this frequency is inferred from Q' – global frequency of a cluster i is the cardinality of the set $Q'[i]$ after Filter. The *IEE* iterates over the query image's clusters, present in Q' (line 44); for each of these clusters, scoring will require the cluster's *idf* (c_{idf}) and its frequency in the query image (c_{q_f}). All pairs belonging to the cluster are iterated (lines 47 to 55); each iteration pertaining to an occurrence of the cluster within image id, with frequency f. Scoring for the occurrence is calculated by taking the query image's own frequency (c_{q_f}) as weight for the *TF–IDF* of clusters within image id (line 52).

After S contains scores for all relevant documents, a dictionary S' is generated by ordering scores in descending order and padding or truncating results to ρ (line 58), hiding the real response size. The resulting S' is then returned to the *Client* through the secure channel (lines 59 to 63).

3.5.2 Security Analysis

In this section we will informally analyse the security of the VISEN scheme. As this scheme shares similarities with BISEN, we summarise some topics of the analysis, and refer to Section 3.4.2 when needed. Such is the case with the *Server*, the *Storage Service*, and the *Client-IEE* Channel analysis: their operation remains the same in BISEN and VISEN, and the security guarantees given in BISEN hold equally in VISEN. In the *Storage*

Service the same integrity guarantees remain through the use of authenticated encryption and by storing a *label* in its corresponding *value*. In this section we will focus on VISEN's operation leakage, as other BISEN guarantees are retained.

Leakage of Procedures We base our leakage in function of data sent through and to untrusted components of our system. The protocol being executed is always leaked[13], as accesses to the *Storage Service* reveal it.

In the *codebook generation phase* VISEN leaks the kind of codebook generation performed. Additionally, *TrainKmeans* leaks the number of clusters, since the codebook is sent via network; *TrainOnlineKmeans* leaks the number of images used for training; *GenerateClustersLSH* produces no additional leakage.

In the *Add* protocol the *Client* sends a fixed-width message – as feature extraction is done locally, thus hiding the size of the image in bits; the cardinality of the feature vector set F being padded to an upper bound, controllable by ω. In the *IEE*, we leak whenever the *entry buffer* is full, and the number of entries it holds.

Without padding, the *Remove* protocol would leak the operation between *Client* and *IEE* – since it only sends an identifier through the network; nonetheless, making it indistinguishable from additions in this step is trivial: the *Client* would send a dummy feature vector set F, padded to the same fixed-width message from *Add*. Given the existence of the *entry buffer*, entries regarding removals are held in the *buffer* as with the *Add* protocol; consequently, the protocol also leaks whenever the *buffer* is full. Nonetheless, *Add* and *Remove* protocols are indistinguishable.

In the *Search* protocol the *Client* sends the feature vector set of the query image to the *IEE* – consequently also a fixed-width message. The *IEE* only requests the relevant clusters to the *Storage Service*, producing *access leakage*; as with the *Add* protocol, all clusters could be retrieved, and the irrelevant ones discarded, however to great performance penalty, as it would imply retrieving the full database, even if for immediately discarding most of it in the *IEE*. Additional leakage due to scoring is not produced, and fixed-width responses give the same leakage as that of ranked BISEN (Section 3.4.3).

In comparison with the state-of-the-art (Table 2.1 from Section 2.3.4), VISEN only produces *access leakage*, as an adversary cannot distinguish whether two queries that produced the same *access pattern* are a repetition of the same query, or two different query images whose features were clustered to the same centroids.

3.5.3 Discussion

While designing VISEN we considered whether to execute all processing at the *IEE* or perform some at the *Client*. We decided to execute feature extraction at the *Client*, since this step produces a fixed number of feature vectors for each image, hence hiding the image's

[13] With *Add* and *Remove* protocols being indistinguishable with the padding we will describe below.

true size and resolution, *i.e.* its detail. This process is akin to the k_w step performed in BISEN (Algorithm 3.2, line 1) to hide keyword length.

The scheme we presented does not consider mutable files, *e.g.* a user cropping or otherwise changing an image. An approach similar to BISEN would incur in the following flaw: since the clusters / keywords of the previous image would not be known (finding them would be computationally expensive for the *IEE*), adding a new image would possibly leave some of the previous version's clusters unchanged. A possible solution would be to delete the image to be changed before re-uploading its new version. Other solution is as follows: a user would upload a new version of the image with the same identifier it had previously, together with a version identifier. This identifier would be stored together with the frequency value in the *Storage*; processing a search would consider the last seen version through a Filter operation similar to that of BISEN. Security guarantees of this version of the scheme would be the same of BISEN, as operations remain indistinguishable outside of the *IEE* and *Client*.

One possible improvement when searching, to avoid requesting all clusters present in the query image, would be to ignore lower frequency clusters. Since the *IEE* knows the frequency of each cluster in the query image (via the image's histogram), it can retrieve only the most frequent, *i.e.* those that will affect the *TF–IDF* metric more significantly. This approach could be based in concepts such as *champion lists* (Manning et al., 2008, Section 7.1.3).

3.6 MISEN: Multimodal Isolated Searchable Encryption

In this section we present our last scheme, MISEN (Multimodal Isolated Searchable ENcryption), a protocol for simultaneous querying of both sparse (*e.g.* text) and dense (*e.g.* images, video or audio) data. MISEN relies on both protocols described earlier, BISEN and VISEN, to perform searches over each data type. MISEN considers the ranked version of BISEN and VISEN, both providing ranked results.

We present a simplified architecture of MISEN in Figure 3.2. Since BISEN and VISEN searches expose the same interface – a query as input, and a list of documents ordered by ranking as output, MISEN's first approach is to treat both schemes as black boxes. Documents of text and image data are sent by the *Client* and added to both schemes separately in the *IEE* (having the same identifier in either scheme). Searches are issued by the *Client*, and may contain both Boolean text queries and query images; the *IEE* processes these queries using a *Request Processor*, which redirects data to BISEN or VISEN as necessary. Both schemes execute their unchanged *Search* procedure, and return the result to a *Ranking Function* module. This module aggregates and ranks the results through a similarity function for multimodal data, such as ISR (Mourão et al., 2013); this function combines scores from both schemes (BISEN and VISEN), attributing a weight to each, and returns the combined result to the *Client*.

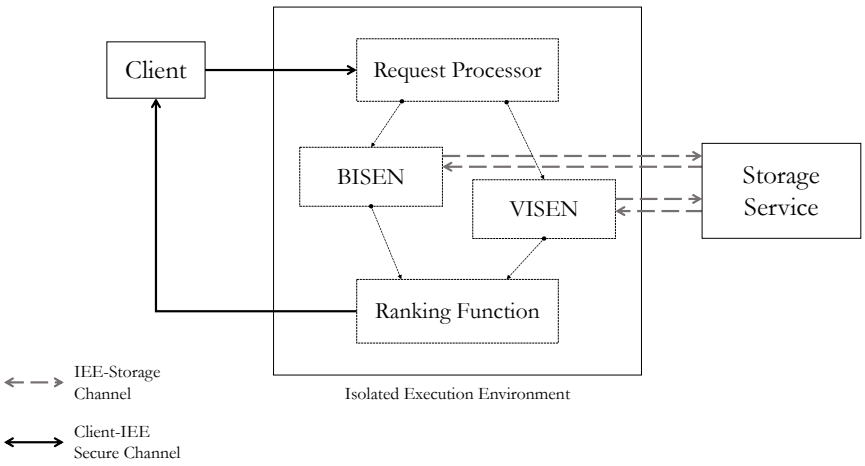


Figure 3.2: Simplified architecture of MISEN, showing internal *IEE* modules and integration with BISEN and VISEN.

By considering some improvements, however, we can provide an enhanced, albeit less modular, version of MISEN. Firstly, the *Storage Service* module could be shared by both schemes. To this effect, data stored on it would need to be annotated to segregate text and image data – which could be done by concatenating an identifier to the counter in the *IEE* label generation steps of BISEN and VISEN.

In the *IEE* we would consider a Storage *Handler* module, responsible for executing *Storage Service* interaction – namely grouping label requests from both parts and randomly permuting them. After requesting the labels and receiving their respective values, the module would decrypt and verify them, delivering an authenticated set of values to each scheme, and thus abstracting the common *Storage Service* interaction they both share.

With these changes, we could parallelise the *Update*/*Add* and *Search* protocols. Processing up to the part of *Storage* requests would be done individually, and values needed from *Storage* would be served from the Storage *Handler*, with further processing again redirected for each scheme, up until results are sent to the *Ranking Function* module.

Chapter 4

Protocol Implementation

In this chapter we will describe the implementation of our searchable encryption schemes, BISEN, VISEN, and MISEN. The Isolated Execution Environment (IEE) described in the protocols is instantiated through Intel SGX (Anati et al., 2013; Hoekstra et al., 2013; McKeen et al., 2013). This instruction set allows us to have a trusted hardware module with confidentiality and integrity guarantees, together with an authentication protocol that enables the IEE requirements from Barbosa et al. (2016).

Our prototype and protocols were implemented in C/C++, using Intel SGX SDK version 2.2[1]. The final prototype amounts to nearly 22 000 lines of code, including the three main prototypes and auxiliary tools for its benchmarking, and is available on https://github.com/sgtpepperpt/MISEN as open-source. Our *Server* and *IEE* modules were implemented as part of a Framework to abstract Intel SGX and its API, while the *Client* and *Storage Service* modules were implemented as separate executables. We implemented versions with *Add* operations (with no update or delete operations) of our three schemes, BISEN (both ranked and exact-match versions), VISEN, and MISEN.

In the remainder of the chapter we will present our Framework for Intel SGX-based IEE abstraction (Section 4.1); detail the implementation of the remaining components, namely *Client* and *Storage Service* (Section 4.2); we then focus on common scheme implementation details (Section 4.3); the chapter concludes with particular details for BISEN (Section 4.4), VISEN (Section 4.5), and MISEN (Section 4.6).

4.1 Framework for Intel SGX

To ease interchangeability and create a common programming interface for our three schemes, we implemented a Framework designed to abstract Intel SGX enclaves as an IEE,

[1] https://software.intel.com/en-us/sgx-sdk

provide common libraries for utilities (such as interaction with the Operating System) and cryptographic functions, and act as a runtime for Intel SGX in the Cloud server. Although in this thesis the Framework is directed at serving our schemes interchangeably, our objective as future work is to develop a Framework to enable a seamless IEE programming environment for generic secure Cloud-based applications.

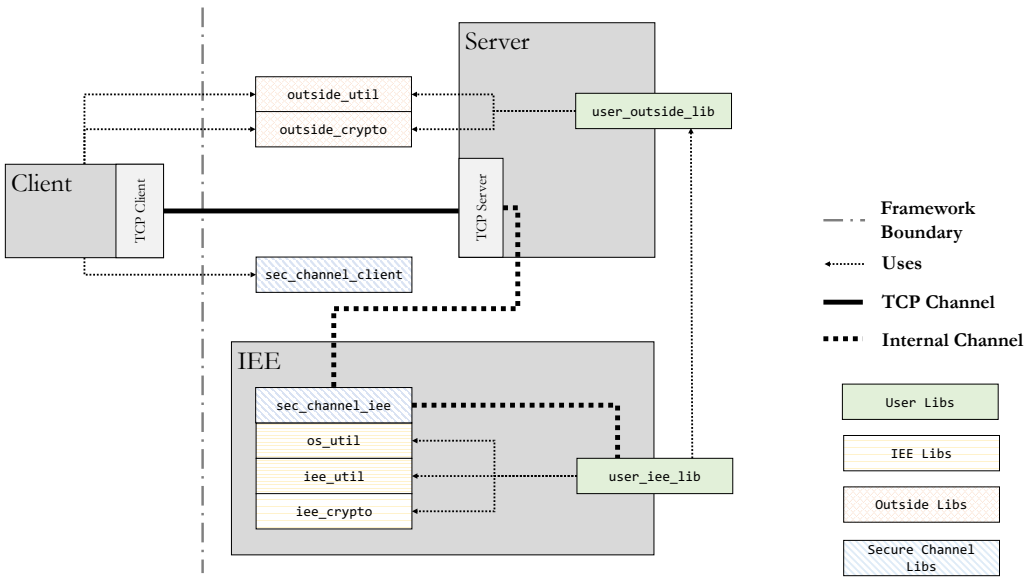


Figure 4.1: Framework architecture. The Framework (right of the boundary) provides a runtime for the *Server* and *IEE*, and a group of libraries as API. User code for the *IEE* and *Server* modules leverages Framework libraries; the *Client*, although outside of the Framework, may also leverage its code.

We present our Framework architecture in Figure 4.1. The Framework's main contribution is to provide a runtime for the *Server* and *IEE* modules in the Cloud server, which are pluggable with user's *IEE* code (and accompanying untrusted code for OS interaction, if needed) via user-programmed static libraries (`User Libs`). To ease *IEE* programming, a group of utility libraries (`IEE Libs`, `Outside Libs`, and `Secure Channel Libs`) is provided as a set of APIs for `User Lib` code.

The runtime for the *Server* acts as host to the *IEE*, and as proxy between *Client* and *IEE*[2], exposing a TCP server and forwarding encrypted messages from the *Client-IEE* secure channel to the *IEE*'s `sec_channel_iee` library, which decrypts and delivers messages to the user library (`user_iee_lib`) for application-specific processing.

Application code for `user_iee_lib` may leverage functions from utility `IEE Libs`, which, apart from fully trusted code (`iee_util` and `iee_crypto`), also provide common OS interaction primitives via `os_util`; if needed, further interaction can be user-defined in `user_outside_lib`.

Some of the Framework's utility libraries are also leveraged by the *Client* module

[2]We omit the *Storage Service* for simplicity, as it does not leverage Framework utilities.

(*e.g.*, for the client-side part of the secure channel establishment). Yet, since our Framework's main purpose is to abstract IEE implementation details, we consider such uses to be orthogonal to this section, and only discuss them where relevant.

Although our Framework was designed so as to provide a generic IEE abstraction for programers, our current implementation only supports Intel SGX, and we leave support for ARM TrustZone as future work. As such, we will use the terms IEE and enclave interchangeably in this section.

In the remainder of the section we describe the initialisation of the Framework (Section 4.1.1), define its API for *IEE* and outside user code (Section 4.1.2), and present a summary of existing utility libraries (Section 4.1.3).

4.1.1 Framework Initialisation

After the user programs their code as a static library – in our case, the *IEE* part of our isolated SSE protocols – and links it to the Framework, the Framework is executed via the following steps:

1. Initialise and start running the Intel SGX enclave, loading the user-provided static library.
2. Wait for a *Client* connection.
3. After a *Client* connects, start redirecting traffic to the *IEE*.
4. The *Client* and *IEE* establish their secure channel (*IEE* and *Client-IEE* Channel Initialisation Procedure, described in Section 3.3).
5. Messages received by the *Server* module are redirected to the *IEE* and processed by the *IEE* library.

During operation, the *Client* sends encrypted messages to the Cloud server using `SecureMsg` primitives (which we describe later in Section 4.1.3); the outside part of the Framework delivers them to *IEE*; the *IEE* part of the Framework, being an endpoint of the secure channel, decrypts, authenticates, and validates such messages, finally delivering them to the user application library, which processes them as needed; the *IEE*'s response being returned back through the secure channel to the *Client*.

4.1.2 Framework API

Programming the user code involves creating static libraries of both *IEE* and Outside request processors (`User Libs`), which must implement the functions from Listing 4.1. The *IEE* part is implemented with the `user_iee_lib::process_message` function, which processes arbitrary messages (via the `in` argument) received through the secure channel, and returns the respective response via the `out` argument. The `sec_channel_iee` then

```
1 namespace user_iee_lib {
2     void process_message(void** out, size_t* out_len, const void* in, const
          size_t in_len);
3 }
4
5 namespace user_outside_lib {
6     void process_message(void** out, size_t* out_len, const void* in, const
          size_t in_len);
7 }
```

Listing 4.1: Framework API for application-specific code.

channels that response back through the *Server*, and then onwards to the *Client*. While inside the `user_iee_lib::process_message` function, the programer's code is guaranteed to be executed inside the *IEE*, except for `os_util` calls.

Since the user code might need to access additional external resources – as is the case with our protocols – we provide an outside processing function `user_outside_lib::process_message`, similar to the previous one, but for execution outside the *IEE*, in the *Server* module. This function allows the programmer to also execute arbitrary code outside of the *IEE*, and can be called from inside the *IEE* via `os_util::process_outside`; such call temporarily exits the enclave via an Intel SGX OCALL. While outside, the message sent from the enclave is processed by the `user_outside_lib::process_message` function as defined by the user, and an answer is returned to the enclave via the `out` argument.

4.1.3 Utility Libraries

The Intel SGX SDK API does not include Operating System calls, since the OS falls out of the trust base, both in our IEE model and Intel SGX's one. However, to implement our protocols, we require, for example, access to the *Storage Service* (via the **Storage** group of calls of our protocols), which has to be mediated by the OS, as the *Storage Service* is outside the *IEE*. To this effect, and to provide useful debugging and benchmarking tools, we implemented a group of libraries – which are to be made available mainly for user code running inside the functions of Listing 4.1 – for both *IEE* internal code, and *normal*, possibly untrusted, outside of *IEE* code[3].

Libraries for *IEE* Code Libraries to be used in the *IEE* are divided into `os_util`, which provides the *IEE* with functions that require OS intervention[4], `iee_util`, with generic *IEE*

[3] In fact, the code implemented by these libraries could be user-implemented via the primitives from Listing 4.1; we provide it to further ease the programming interface of the library.

[4] `os_util` library functions are not completely trusted, as data from its arguments is both passed to and originated from outside the trust base. Since the library is to be used from inside the *IEE*, we choose to include it alongside other such libraries. Security of data manipulated or obtained via such functions has to be ensured by the programmer.

```
namespace os_util {
    // file i/o
    int open(const char* filename, int mode);
    ssize_t read(int file, void* buf, size_t len);
    ssize_t write(const int file, const void* buf, const size_t len);
    void close(int file);

    // tcp communication
    int open_socket(const char* addr, int port);
    void socket_send(int socket, const void* buff, size_t len);
    void socket_receive(int socket, void* buff, size_t len);
    void close_socket(const int socket);

    // interaction with storage service
    void storage_message(const int socket, void** out, size_t* out_len, const
        void* in, const size_t in_len);

    // outside allocation
    void* outside_malloc(size_t length);
    void outside_free(void* pointer);

    // generic calls
    void printf(const char* fmt, ...);
    void exit(int status);
    time_t curr_time();

    // implemented by user_untrusted_lib::process_message, generic untrusted
        processing
    int process_outside(void** out, size_t* out_len, const void* in, const
        size_t in_len);
}
```

Listing 4.2: *IEE* library `os_util`.

trusted functions, and `iee_crypto`, with *IEE* cryptographic functions. We now describe each library's purpose, and guide the reader to Appendix A for complete library APIs:

- `os_util` (Listing 4.2): abstractions for OCALLs, such as insecure file I/O operations, TCP communication (to establish channels and communicate with generic external modules), a *Storage Service* abstraction for message sending and receiving, abstractions for memory allocation outside the enclave (needed when passing variable-length data to and from the outside code), generic functions for debugging purposes, and a `process_outside` call to cover any other necessary functionality – to be implemented by `user_outside_lib ::process_message`.

- `iee_util` (Listing A.1): implements a thread pool for parallelism within the enclave[5], and provides secure file I/O operations using data sealing (see Section 2.4.3.1).

[5]Threads cannot be instantiated inside the enclave, but parallelism is possible via a thread pool. See https://github.com/intel/linux-sgx/issues/106.

- `iee_crypto` (Listing A.2): contains necessary cryptographic functions. In our protocols, we require two cryptographic functions, a PRF F and an authenticated symmetric cipher Θ – these were instantiated as a SHA256-HMAC, and XSalsa20 stream cipher with Poly1305 MAC, respectively. We leveraged the implementation of LibSodium[6] for these functions, as it contains built-in authentication mechanisms[7]. For the random functions we used the source of randomness provided by Intel SGX SDK.

Libraries for Outside Code

- `outside_util` (Listing A.3): contains functions for debugging and TCP communication, which can be useful to establish communication with other system components – and are also leveraged by functions such as `os_util`'s TCP functions.
- `outside_crypto`: mimics the same interface and behaviour of `iee_crypto` for untrusted parts of the code; we therefore omit it for brevity. Its main purpose is to provide a cohesive cryptographic interface for outside of *IEE* operations, such as those done by the *Client*.

Libraries for Secure Channel Communication

- `sec_channel_client` (Listing A.4): functions for the *Client* to respectively establish, use and terminate a secure channel connection to the *IEE*, abstracting TCP communication with the *Server*.
- `sec_channel_iee` (Listing A.4): functions for the *IEE* to control secure channel connections. The Framework initialises a secure channel server in the *IEE*; when a *Client* connects the authentication protocol is performed, and further messages are delivered from this internal server into the user application library.

4.2 Implementing the *Client* and *Storage Service*

The *Storage Service* only requires a key-value store interface, with *put* and *get* operations. We used a memory-efficient hash map implementation – Sparsepp[8] – with a similar interface to that of *map* from the C++ standard library, with additional alternative implementations in Redis[9] and Cassandra[10]. The *Storage Service* is exposed as a TCP server, and receives messages from the *IEE* via its `os_util::storage_message` function, to which the *Storage* interprets and replies as needed. For memory allocation, the *Storage Service* has to be aware of the size of the $(key, value)$ pairs it receives; in either protocol, key is

[6] https://libsodium.org/
[7] See https://download.libsodium.org/doc/secret-key_cryptography/authenticated_encryption.
[8] https://github.com/greg7mdp/sparsepp
[9] https://redis.io/
[10] https://cassandra.apache.org/

32 bytes long – a SHA256-HMAC, and *value* is 80 bytes long – 32 bytes from the *key* validation, 4 bytes to represent the document / image identifier, 4 bytes for the frequency value, and 40 bytes for LibSodium's authenticated cipher data (a nonce and a MAC).

Our *Client* acts as a TCP client, which implements our schemes' client-side logic, along with the communication protocol needed for the encoding and decoding of *IEE* messages. Moreover, it implements the secure channel logic needed to authenticate and communicate with the *IEE* via `sec_channel_client`, and leverages cryptographic primitives from our `outside_crypto` library.

4.3 Common Implementation Details

In this section we briefly describe some common implementation details of our protocols not included with the Framework description, namely how common patterns found on the schemes are translated into implementation.

Communication (namely `BISEN` and `VISEN` calls used in the *Client* part of our protocols) is done by generating binary messages which contain the necessary data to be transmitted to the *IEE*. These are sent abstracted through the `sec_channel_client` functions from the Framework, while the **Storage** group of calls use the `os_util::storage_message` function. While the protocols show primitives for single pair put/get operations, values sent to and from the *Storage Service* are batched for communication efficiency.

Data structures, such as dictionaries and sets, used throughout our protocols were implemented as C++ standard library `maps` or `vectors`. In particular cases, such as with BISEN, we used custom vector data structures to help with set processing operations – for example, having ordered document sets while resolving queries facilitates algorithm optimisations, not only performance but also memory-wise.

4.4 Implementation of BISEN

In this section we detail implementation decisions taken for BISEN, in particular for the *Update* and *Search* operations.

Update Operation Our implementation of the *Update* operation takes raw text documents as input. The *Client* executes a preprocessing step, with techniques from Manning et al. (2008, Section 2.2), which are as follows: a word index (word associated to its frequency) is initialised; for each word of the document, we check whether it is a stop word (*i.e.*, a common word with little semantic value, such as *the* or *and*), and ignore it – if not, we add the morphological root of that word to the index (known as stemming), or increment its frequency if already present. The set of all words of a document is sent in batch to the *IEE*. While the current implementation approach leaks document size – as each update is a document addition, the implementation could easily be changed to send

fixed-width messages, holding parts of the document until an amount of updates is ready to be sent in batch, or padding messages to a fixed size.

Search The *Search* operation takes an ASCII text query as input, and the *Client* executes the same preprocessing of *Update* for each word. Document sorting, if scoring is enabled, is done via the Quicksort implementation available in C.

4.5 Implementation of VISEN

VISEN's image-related operations rely on version 3.4.1 of the OpenCV[11] library. Particularly, we used both its SURF and SIFT implementations for feature extraction[12], and its k-means implementation for the *Traditional K-means* approach.

Feature extraction of an image results in a set of `float` vectors, with 128 dimensions each for SIFT, and 64 for SURF. Clustering data into a codebook of k clusters results in a set of k vectors of `float`, which are ultimately stored in the *IEE*. Apart from this, the initialisation and communication protocol of VISEN is similar to that of BISEN.

Codebook Generation Phase We implemented the three versions for codebook generation referred in Section 3.5.1. The *Traditional K-means* approach was implemented at the *Client*-side via the OpenCV class `BOWKMeansTrainer`; the *Online K-means* was implemented by us following a simple algorithm for online cluster updating[13], with the seeding phase corresponding to the first vectors sent to the *IEE*; finally, clusters for *LSH* are generated using random `float` values following a normal distribution.

Use of Parallelism Some procedures of VISEN's protocols are embarrassingly parallel. That is the case of approximation procedures (when reducing feature vectors into clusters), where each vector's calculation is totally independent from others; and the case of addition of data to the *Storage Service*, where putting and getting each label does not require any type of ordering. We implemented both sequential and parallel versions of these procedures.

4.6 Implementation of MISEN

MISEN was implemented by grouping BISEN and VISEN functions into the same static library, creating an additional request processor in front of these schemes, to separate queries for either one. Our implementation did not use the same *Storage Service* for both schemes, instead relying on two instances of it. Both schemes operate sequentially, in part since they share some Framework resources, and also due to Intel SGX limitations on

[11] https://opencv.org/
[12] Available as an extra module at https://github.com/opencv/opencv_contrib.
[13] https://www.cs.princeton.edu/courses/archive/fall08/cos436/Duda/C/sk_means.htm

parallel programming: since VISEN uses parallel functions, and SGX threads are limited to physical cores, VISEN's operation could not be trivially parallelised with BISEN's own operation, and, consequently, we leave such parallelism as future implementation work.

Chapter 5

Experimental Evaluation

In this chapter we present the experimental evaluation performed over our schemes. Our first objective was to assess the latency and scalability of each of their operations, and the impact each of the system's participants (*Client*, *IEE* and *Storage Service*) had on overall latency. We also discuss the impact of some implementation decisions and alternative approaches for parts of our schemes. Furthermore, we compare our schemes with the state-of-the-art on text (Kamara and Moataz, 2017) and image (Ferreira et al., 2018) SSE schemes.

In the remainder of the section we describe our experimental test bench (Section 5.1), and the results obtained for BISEN (Section 5.2), VISEN (Section 5.3), and MISEN (Section 5.4).

5.1 Experimental Test Bench

Our experimental test bench was composed of an SGX-enabled machine, and an external server with large memory to act as *Storage Service*. Our *Client*, *Server* and *IEE* were deployed on the same machine for practical reasons.

Our SGX-enabled machine was a 4-core Intel NUC i3-7100U, with 2.4GHz of CPU frequency, 8GB of RAM and 256GB of SSD storage, running *Ubuntu Server 18.04.1*. Our server for the *Storage Service* contained an AMD Opteron 6272 64-core CPU with 2.1GHz of frequency, 64GB of RAM and 128GB of SSD storage, running *Ubuntu Server 16.04.4*. Both machines were deployed in a local network – nevertheless, to avoid network noise, we consider local latencies of the main participants individually (*Client*, *IEE* and *Storage Service*), but not network latency (we still consider the time of buffer allocation for messages, and their respective writing and reading). We omit *Server* performance, as its work consists solely of message forwarding, and so we considered it as network time.

5.2 BISEN Evaluation

BISEN's evaluation is centred in its *Update* and *Search* protocols. We evaluated BISEN using the English Wikipedia Dataset (Wikipedia, 2018) from August 2018, which contains around 5.5 million articles (*i.e.* documents), and amounts to ≈ 60GB of raw data. To cleanup metadata (such as Wikipedia templates, links, and tables) we used a parser tool, WikiExtractor[1], which yields pure text documents, amounting to ≈ 13GB of text data. Due to memory constraints, our *Storage Service* was unable to support the full dataset in-memory; as such, we use about 5 million articles of this dataset in our experimental evaluation, amounting to about 464 million entries on the *Storage Service* index.

Our searches were performed by combining the twelve most popular words from the English language, which are, from most to least popular: *time*, *person*, *year*, *way*, *day*, *thing*, *man*, *world*, *life*, *hand*, *part*, and *child*. We joined these words in different types of queries, as presented in Appendix B, using Boolean conjunctions (&&), disjunctions (||), negations (!) and sets of parentheses. Due to the high impact of the scoring function, our results present the unranked version of BISEN, except for Section 5.2.7, where scoring is discussed.

Our tests start by presenting the latency of both *Update* and *Search* protocols, and the impact of each protocol participant individually (Section 5.2.1). To evaluate the *Search* protocol we analyse latency with different types of Boolean operators and operands (Section 5.2.2), the impact of different operations in *IEE* processing (Section 5.2.3), the impact of the number of negation operators in queries (Section 5.2.4), and conclude the *Search* protocol analysis with an assessment of latency under varying keyword selectivity (Section 5.2.5).

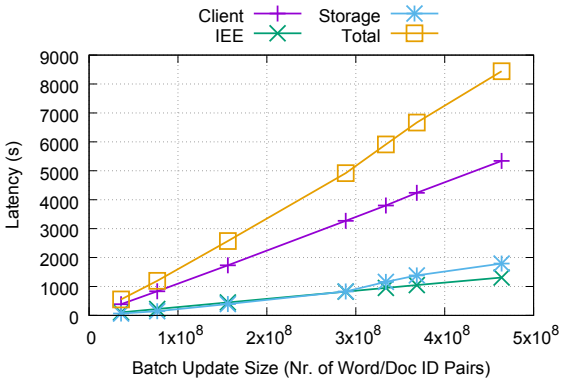
In the remain of the section we compare three different solutions for the *Storage Service* (Section 5.2.6), and assess the overall performance impact of scoring in ranked BISEN (Section 5.2.7). We conclude by comparing our scheme with the state-of-the-art from Kamara and Moataz (2017) (Section 5.2.8) and discussing obtained results (Section 5.2.9).

Except if otherwise noted, our latency plots will show the database size (*i.e.*, the number of word/document id pairs in the *Storage*) in the x-axis, and the latency of operations in the y-axis. Latency of these operations was measured at increasingly larger database sizes – from 500 thousand articles up to 5 million (90% of Wikipedia, the maximum value whose index fitted the 64GB of RAM of our *Storage Service* machine).

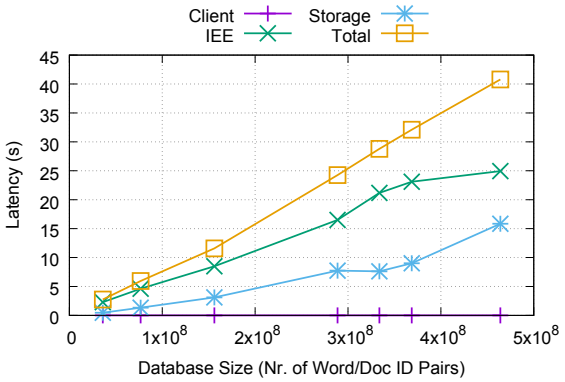
5.2.1 Performance of Individual Participants

In this test (Figure 5.1) we assess the impact of each participant (*Client*, *IEE* and *Storage Service*), and the total latency, in both *Update* and *Search* protocols. We observed updates (Figure 5.1a), performed in batches of increasingly larger sizes, scale linearly to that size, which implies single *Update* operations are constant. Time spent in the *IEE* and *Storage* is

[1] https://github.com/attardi/wikiextractor



a BISEN *Update* protocol.



b BISEN *Search* protocol, for an example conjunctive query of five keywords.

Figure 5.1: Performance comparison of each participant for the *Update* and *Search* protocols of BISEN.

roughly similar, with a tendency for the *Storage Service* to become a bottleneck for larger operations. While we consider a single server, distributing the *Storage Service* across multiple machines, and parallelising update batches – even if without replication and availability guarantees – might mitigate its weight in the operation. In turn, the *IEE* is responsible for simple cryptographic computations, which is reflected in the latency for the maximum update (1300 seconds for 90% of Wikipedia). The largest slice of *Update* processing is on the *Client* – as we account for pre-processing of documents (described in Section 4.4), which take about 46% of the shown *Client* time. Reading files from disc accounts for 2% of the time, while the rest is spent in counter operations and buffer allocation and writing.

Regarding *Search* (Figure 5.1b), most of the processing is done in the *IEE* – this is due mainly to buffer manipulation (producing the labels for *Storage* requests, and processing and storing responses inside the *IEE*; we further discuss this result in Section 5.2.3). Contrarily to the *Update* protocol, the *Client* has almost no processing latency, as it only has to pre-process the query keywords, and not full documents. As expected, we observed searches scale linearly to database size – with a larger database, a keyword is expected to be present in more documents, all of which have to be retrieved into the *IEE*[2].

5.2.2 Performance Regarding Type of Query

With this test (Figure 5.2) we wanted to assess the impact of both the type of operators and the length of the query on overall latency. We used queries in both Conjunctive Normal Form (CNF) and Disjunctive Normal Form (DNF), with one, three and five conjunctions and disjunctions. These correspond, for example, to queries of the form $(A \vee B) \wedge (C \vee D)$ (one conjunction) or $(A \vee B) \wedge (C \vee D) \wedge (E \vee F) \wedge (G \vee H)$ (three conjunctions) for the CNF; the same logic applies to the DNF. The queries executed are 24-29 from Appendix B.

[2]This is directly related with keyword selectivity, which we discuss in Section 5.2.5.

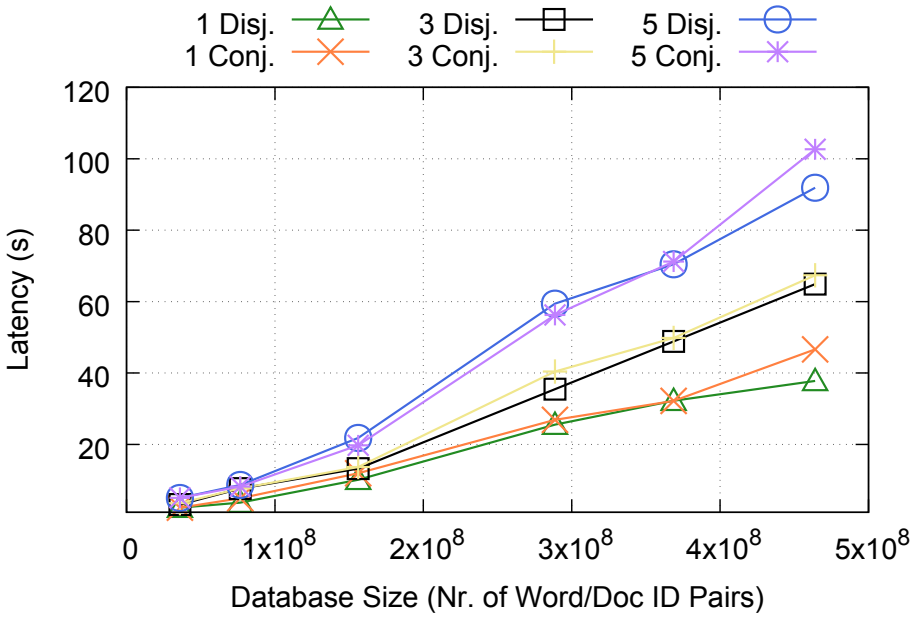


Figure 5.2: Impact of the Boolean formula and query size on the performance of the BISEN *Search* protocol. The plot key describes the number of the main operator of each query (*e.g.*, *1 Disj.* represents a query in Disjunctive Normal Form (DNF) with a single OR.)

Our main takeaway from this experiment is that the type of Boolean operator does not significantly affect query performance (*i.e.*, the impact of resolving AND and OR operations is similar), while the number of operands does – retrieving 12 keywords (*3 Disj.* and *3 Conj.* in the figure) incurs in a bigger latency than retrieving 8 or 4 keywords, as the number of entries to be retrieved by the *IEE* increases accordingly.

5.2.3 Impact of *IEE*-specific Operations during *Search*

In this section we briefly discuss the impact of different operations in IEE processing for the *Search*. In ranked BISEN, scoring takes the larger slice of processing, with an average of 85% processing time dedicated to scoring (via *TF–IDF*) and sorting of the resulting document list using Quicksort (which we further analyse in Section 5.2.7).

Excluding scoring, Boolean set processing takes a minimal amount of time (averaging 2% of the time, and peaking at 6% for more complex queries), while most of the time is spent in buffer copying and allocation, such as message generation for the *Storage Service* (≈76% of time), its respective decoding (≈20%), and preparation of data structures for Boolean processing (≈2%). Therefore, we conclude the bottleneck in processing is due to interaction with the *Storage Service*, which takes ≈96% of time, while Boolean processing in itself is a light operation.

DB Size (Nr. of pairs word / doc id)	**1 Negation**	**5 Negations**	**10 Negations**	**Fully Neg.**	**De Morgan of Fully Neg.**
35 996 207	4.286	4.498	3.052	4.319	3.565
76 672 004	9.335	9.241	9.610	7.185	9.041
156 143 147	18.653	18.092	21.095	16.589	18.379
288 706 216	55.823	55.461	52.758	56.768	54.946
333 784 724	52.265	58.227	50.850	51.996	57.290
368 651 490	66.277	67.845	66.048	64.580	65.956
464 054 543	86.057	82.289	85.041	86.938	88.563

Table 5.1: Performance of negations in the BISEN *Search* protocol, with time in seconds. Queries shown are, respectively, 30-34 from Appendix B.

5.2.4 Performance of Negation Queries

In Table 5.1 we present the impact of negations for queries of fixed size (10 keywords), varying the number of negated keywords – one, five and ten; then a fully negated query – of the form $\neg(A \wedge B)$, and finally the equivalent version of the latter using De Morgan's laws. Our objective was to assess the impact performance of the negation operation across different types of queries and numbers of negations. Results show that the number of negation operations performed has minimal impact, even for larger database sizes, which can be explained by the low overhead of Boolean processing. Since all queries require the same number of entries to be fetched from *Storage*, where the main bottleneck is, their latency is, therefore, similar.

5.2.5 Performance Regarding Selectivity

In this experiment we analyse the impact of keyword selectivity in the *Search* protocol – *i.e.*, how many documents each keyword appears in; higher selectivity implies more requests to the *Storage Service*, and larger responses to the *Client* in the unranked version. We performed single-keyword queries returning from ≈0.2% of the database, up to ≈25% of all documents.

Results show (Figure 5.3) that an increase in latency is linear with selectivity, with a tendency to be amortised in larger databases. Taking into account the significant impact of *Storage Service* interaction on overall latency (Section 5.2.3), and considering that increasing selectivity implies more entries to be requested and processed by the *IEE*, the linear increase in latency is expected.

In conclusion, selectivity of keywords is the factor which determines overall *Search* latency on exact-match BISEN, as it determines the number of entries that have to be requested by the *IEE*; with a single entry request incurring in constant time (processing one entry on the *IEE*, and a single access in the *Storage Service* dictionary), a linear increase in requests incurs in linear time.

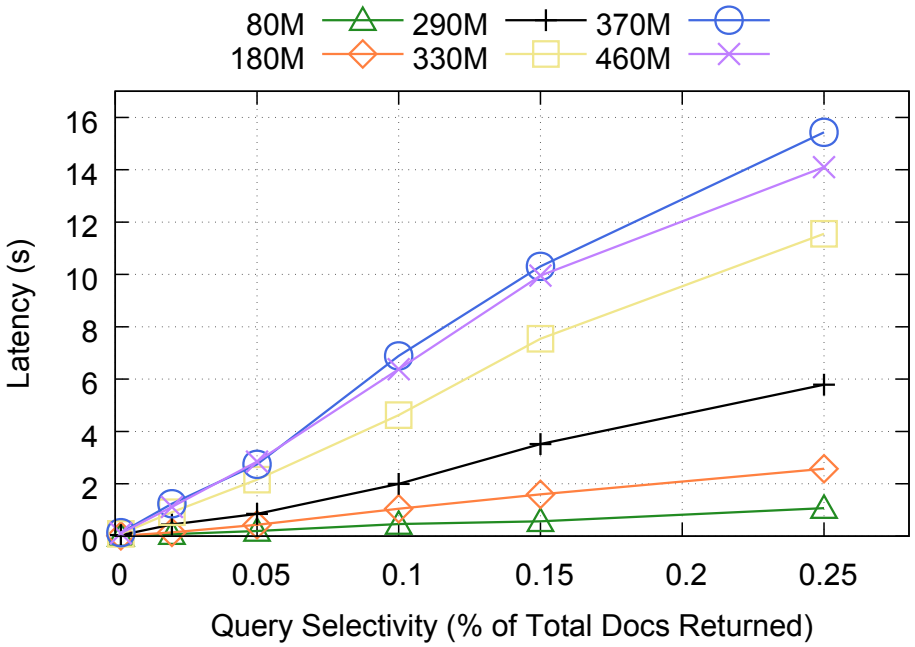
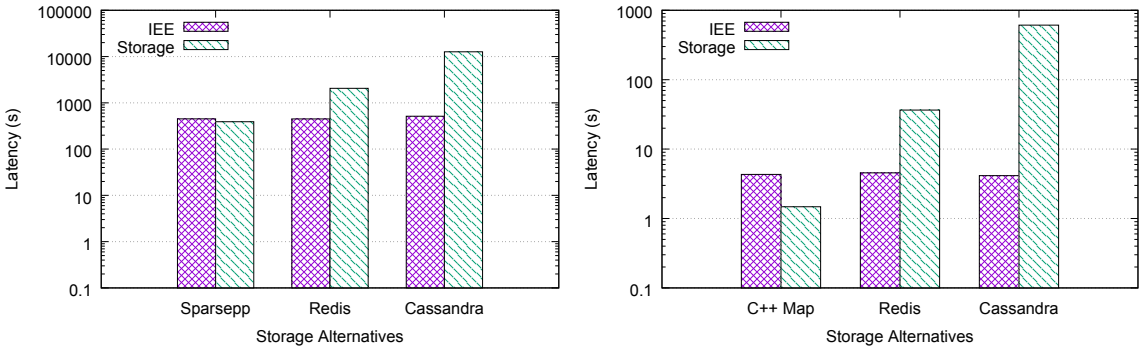


Figure 5.3: Impact of query selectivity on the performance of the BISEN *Search* protocol. The y-axis shows latency for increasing selectivity on the x-axis, with samples for different database sizes.

5.2.6 Evaluating Different *Storage* Solutions



a BISEN *Update* protocol.

b BISEN *Search* protocol, for a conjunctive query of two keywords.

Figure 5.4: Comparison of BISEN performance with three different solutions for *Storage*: *Sparsepp*, *Redis* and *Cassandra*. Logarithmic scale used due to the difference in orders of magnitude between solutions.

Our main implementation for the *Storage Service* uses Sparsepp, an in-memory map optimisation of the `unordered_map` from the C++ standard library. However, as it might present itself as an *ad-hoc* solution with no support for persistency and fault tolerance, we implemented drivers for two currently popular NoSQL databases – Redis and Cassandra. We compare these two NoSQL solutions in single-node clusters with Sparsepp using a fixed database size of ≈156M pairs (two million documents), and account for the latency

of both *IEE* and *Storage Service* in the protocols.

In Figure 5.4 we compare the three solutions in the *Update* and *Search* protocols. While time spent in the *IEE* is kept unchanged across solutions (as different *Storage* approaches are transparent to it); performance in the *Storage Service* for the Redis approach incurs in a performance penalty of an order of magnitude above Sparsepp, and Cassandra a further order of magnitude above, in both protocols. While Cassandra uses disc storage, which we expected to be a significative hindrance on performance, Redis is an in-memory store, and thus we assumed its performance would be similar to that of Sparsepp.

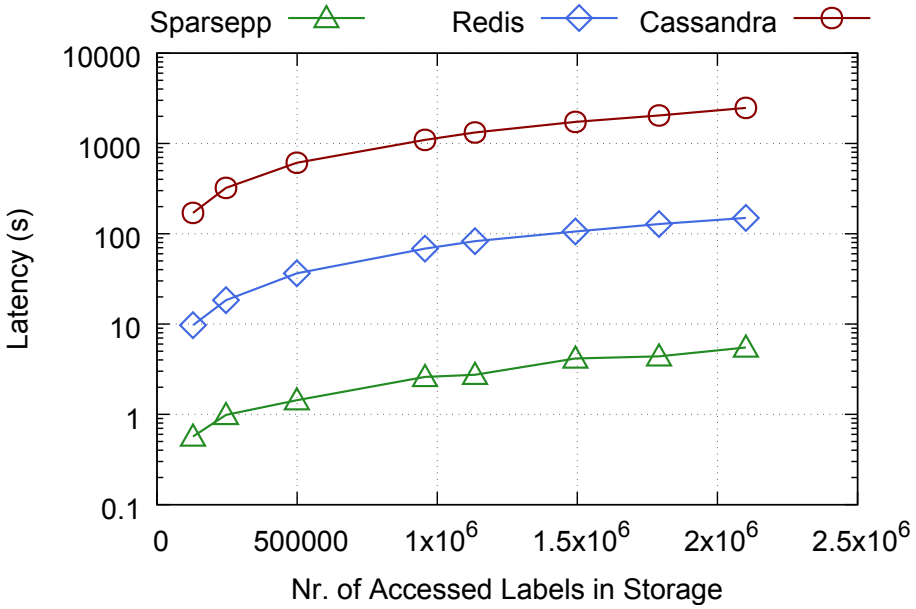


Figure 5.5: Comparative impact of number of accessed entries in the scalability of BISEN *Storage Service* solutions. Logarithmic scale used due to the difference in orders of magnitude between solutions.

Although *Storage Service* latency remains linear to the number of accessed labels (Figure 5.5) – thus presenting NoSQL approaches as viable, at least in a scalability perspective – the overall impact is still larger than expected in our tests. We leave testing the NoSQL databases with more than one node – thus parallelising costly accesses to disc across different nodes – as future work. However, and given Intel CPUs with SGX are limited to eight cores as of 2018[3], performance improvements might then be bottlenecked on the *IEE* side.

5.2.7 Impact of Scoring Algorithms during *Search*

In this section we analyse the impact of scoring operations in *IEE* processing in BISEN. Figure 5.6 shows latency of IEE processing in exact-match and ranked versions of BISEN, in regards to the original document list retrieved from *Storage*, *i.e.* the length of the document list before any scoring and sorting.

[3]As listed in https://tinyurl.com/sgxlist.

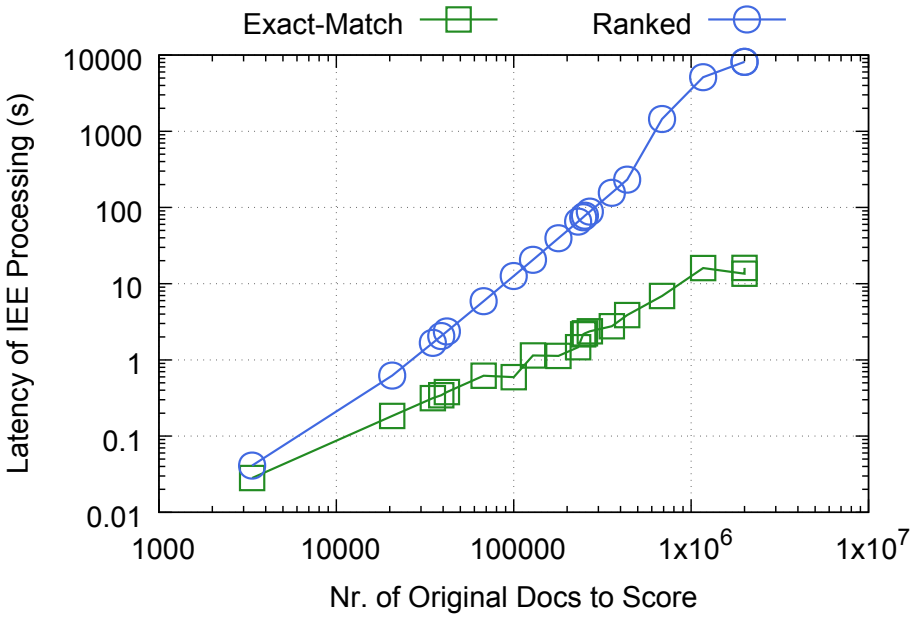


Figure 5.6: Comparison of IEE *Search* processing latency between exact-match and ranked versions of BISEN, in regard to the number of documents retrieved from the *Storage Service*, before scoring, sorting and trimming of that document list. We use logarithmic scales due to the high variation in the orders of magnitude measured.

We fixed the database size at two million articles, and executed queries with different selectivity. We found scoring time to be determined by the number of documents to score, *i.e.* the documents originally retrieved by the query from *Storage* (selectivity). Hence, we chose to plot latency in function of selectivity for better clarity, instead of varying database size while executing the same query, as with former plots. Regardless, having the plot in function of database size would produce the same typology of results, since a given query's selectivity is constant across database sizes (*i.e.*, the same query will always retrieve the same percentage of documents, regardless of database size, with a measured standard deviation of 0.6% in our experiments).

Our results show that the exact-match version of BISEN *IEE* processing grows linearly, and does not exceed 10 seconds for the processing of the two million documents. In the ranked version, the impact of scoring is particularly evident for document sets with larger cardinality, *e.g.* for two million documents processing time increases three orders of magnitude in contrast to the unranked version. In fact, the sorting algorithm used, Quicksort, has $O(n \log n)$ time complexity, which explains the approximately linear behaviour of our results. However, it still prohibitively impacts *Search* performance, taking about 85% of processing time, which indicates the need for further optimisation and research on alternative, more efficient, approaches for the problem of scoring and sorting documents in the *IEE*.

Database Size (Nr. of pairs word / doc id)	Update		Search CNF	
	BISEN	**IEX-2LEV**	**BISEN**	**IEX-2LEV**
9 793	0.151	5 143	0.004	12
27 446	0.423	15 568	0.021	173
56 238	0.862	29 274	0.061	216

Table 5.2: Latency comparison between BISEN and IEX-2LEV (Kamara and Moataz, 2017) schemes. All times are in seconds, queries composed of eight keywords.

5.2.8 Comparison with the State-of-the-Art

To compare BISEN with the state-of-the-art (Table 5.2) we used the implementation of IEX-2LEV from Kamara and Moataz (2017)[4]. Due to space requirements of the IEX-2LEV scheme, we used our AMD Opteron server with 64GB of RAM for these tests. Yet, we were unable to run the scheme with more than 56 238 index entries; for our scheme, we used a software simulation of Intel SGX, so as to compare both solutions on the same machine.

From these results, we conclude BISEN is much more efficient than IEX-2LEV: while our scheme only takes 0.862 seconds to perform a full database update, IEX-2LEV takes more than 8 hours. Our performance advantage is due to the use of trusted hardware – its security guarantees allow us to use simpler and more efficient data structures without compromising security, whereas IEX-2LEV requires quadratic storage, thus penalising both storage and performance considerations.

5.2.9 Discussion

Our evaluation of BISEN allowed us to assess the scalability and performance limits of the implemented solution. The problem of scalability, which the current state-of-the-art leaves as open research, is improved by our solution, which is shown to scale linearly, in both time and storage dimensions.

Such fact makes our scheme predictable in its scalability and performance. By using complementary techniques, such as keeping caches of popular documents in the *IEE*, and distributing load across multiple devices, we consider that our solution might easily become more practical.

However, two features which improved our work's security and dependability guarantees, ranking results and using persistent databases, respectively, still incur in high performance costs. The limiting factor in scoring is the number of documents to be scored and sorted; providing better alternatives that still leverage the advantages of scoring (more significant results and better security) is an interesting line of future work. A first approach could be based in *champion lists* (Manning et al., 2008, Section 7.1.3), which could keep pre-computed scores of documents that contain popular keywords in

[4]Available on https://github.com/encryptedsystems/Clusion.

memory (in our case, inside the *IEE*). When issuing a query containing such popular keywords, the score of those documents would be, at least, partially calculated. Updates to the database would need to be taken into account for the pre-computed scores, either immediately, or periodically, to the cost of temporary loss of precision.

Finally, the results for different *Storage* solutions, and the high RAM requirements of the in-memory Sparsepp solution, show us that an approach combining a fast in-memory solution with the load distribution features of a solution like Redis or Cassandra can achieve a better, more practical approach, which can be explored in future research work.

5.3 VISEN Evaluation

Our VISEN evaluation is focused on assessing not only the scheme's performance and scalability, as with BISEN, but also its precision, Mean Average Precision (mAP), in particular by comparing different training methods. We reinforce that, regardless of the chosen training method, performance of the *operating phase* protocols of VISEN is independent of training (but not of scheme parameterisation, and assuming each method produces centroids following a similar distribution), as either approach outputs a codebook of the same format and size.

To evaluate the *codebook generation phase* of our scheme we used the INRIA Holidays dataset (Jegou et al., 2008), amounting to 1491 images, and which contains built-in precision evaluation tools. To perform scalability and latency tests over the *operating phase* protocols we used the MIR Flickr dataset (Huiskes and Lew, 2008), which contains 25 000 images. We do not present tests particularly targeting the *Storage Service*, as BISEN already covers the module, which is used unchanged for VISEN.

We divide this section in four: in Section 5.3.1 we evaluate VISEN's precision and training algorithms performance; in Section 5.3.2 we analyse the *Add* and *Search* protocols regarding their performance and scalability, under different parameterisations. Both sections compare the effect of the same parameters; however, the former is focused on performance during the codebook generation phase, and resulting precision for the *operating phase*, while the latter section focuses solely on the performance impact such parameterisations have on the *operating phase* protocols. To conclude, in Section 5.3.3 we compare VISEN with the state-of-the-art from Ferreira et al. (2018), and discuss our results in Section 5.3.4.

5.3.1 Evaluating the *Codebook Generation Phase*

With these tests we assessed not only the performance of our three codebook generation approaches (*Traditional K-means*, *Online K-means*, and *Locality-Sensitive Hashing (LSH)*), but also their resulting precision, using the tools provided with the INRIA Holidays dataset. We varied the four possible parameters for codebook generation: the algorithm,

its parameter k (the number of clusters), the feature extraction algorithm, and its parameter ω (which controls the number of feature vectors an image produces).

Available feature extraction algorithms on our implementation were SIFT and SURF. In our experiments we were unable to achieve a mAP larger than 1% for SURF – therefore, our tests focus only on SIFT, which achieves a precision of 49% by combining different ω and k values.

We will present our results by first varying the generation algorithm and its k for a fixed feature extractor (Section 5.3.1.1); we then fix a value for k and vary feature extractor parameters (Section 5.3.1.2). By varying these parameters, we are able to present the best parameterisations possible for our scheme's retrieval precision during normal usage in the *operating phase*.

5.3.1.1 Varying the Number of Clusters

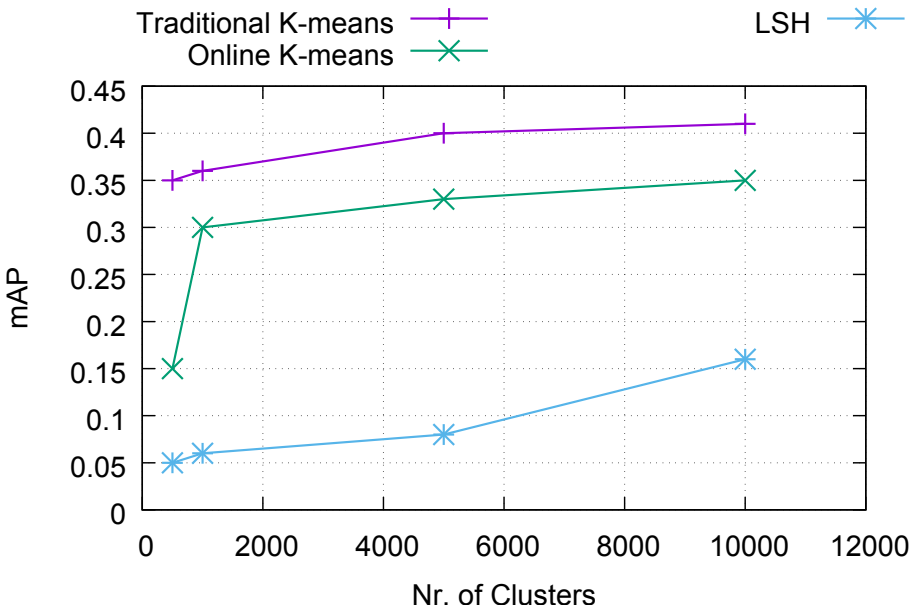


Figure 5.7: Mean Average Precision (mAP) of VISEN under different codebook generation approaches.

We started by varying the number of clusters for each of the three codebook generation algorithms, using the SIFT feature extractor with $\omega = 500$. Using the INRIA Holidays dataset, the feature extractor produced 721 315 feature vectors for the total 1491 images, which were then trained with $k \in [500, 1000, 5000, 10000]$[5].

Figure 5.7 shows our obtained values for mAP, and Table 5.3 presents the respective latencies for this test. Traditional K-means exhibits the best precision throughout, with a maximum of 41%. The result was expected, as this method is the only one to have a

[5]Such values were chosen based on prototypes from the literature (Ferreira et al., 2018); tests with higher k values were not performed due to the high time cost of the training operation, and due to our conclusion that varying ω was more influential on precision, which we discuss in Section 5.3.1.2.

Training Method	$k = 500$	$k = 1000$	$k = 5000$	$k = 10000$
Traditional K-means	9 099 (≈ 2h 32m)	15 398 (≈ 4h 17m)	65 384 (≈ 18h 10m)	127 905 (≈ 1d 12h)
Online K-means	2 962 (≈ 49m)	3 121 (≈ 52m)	3 843 (≈ 1h 4m)	4 839 (≈ 1h 21m)
LSH	0.078	0.103	0.244	0.454

Table 5.3: Performance of VISEN training algorithms with varying number of clusters. Main time in seconds, approximate time also shown.

full vision of the dataset, thus being able to better readjust cluster centroids with several passes per vector; yet, it also grows to almost-prohibitive latency costs for higher number of clusters. *Online K-means* achieved a 35% mAP, a lower value attributable to the algorithm's sensitivity to the order of vectors input, and its single pass over each datapoint – albeit, given its much slower growth in latency compared to the traditional method, still presents itself as a viable approach.

Finally, *LSH* only achieved a precision of 16%. Still, by consisting only of a vector-generation procedure, and requiring no training *per se*, its latency is several orders of magnitude below that of the other approaches (always under one second), and its latency depends solely on k, further requiring no training dataset.

5.3.1.2 Varying Feature Extractor Parameters

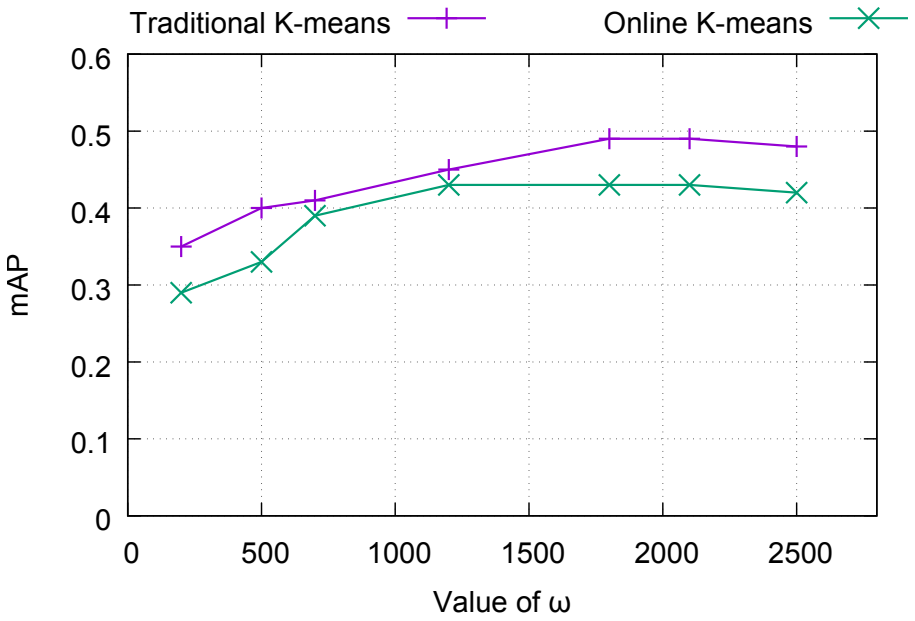


Figure 5.8: Mean Average Precision (mAP) of VISEN under different values for ω.

In this test we fixed $k = 5000$, and performed codebook generation for $\omega \in [200, 500, 700, 1200, 1800,$

2100, 2500]. As with the previous test, we used the INRIA Holidays dataset. The number of vectors used for codebook generation varies with ω, which represents an upper bound on the number of vectors each images produces; thus, each image produces, on average, a lower number of vectors, *e.g.* with $\omega = 200$ each image averages 197 feature vectors, and with $\omega = 2500$ averages 2271 vectors. Regardless, increasing ω represents a proportional increase in the number of vectors used for training. Due to the low mAP attained with *LSH*, we only use the two k-means methods in this test.

Training Method	$\omega = 200$	$\omega = 500$	$\omega = 700$	$\omega = 1200$	$\omega = 1800$	$\omega = 2100$	$\omega = 2500$
Traditional K-means	24 305 (≈ 6h 45m)	65 384 (≈ 18h 10m)	89 183 (≈ 1d 1h)	157 580 (≈ 1d 20h)	235 773 (≈ 2d 17h)	274 869 (≈ 3d 4h)	326 998 (≈ 3d 19h)
Online K-means	3 233 (≈ 54m)	3 843 (≈ 1h 4m)	4 377 (≈ 1h 13m)	5 259 (≈ 1h 28m)	6 511 (≈ 1h 48m)	7 053 (≈ 1h 58m)	7 919 (≈ 2h 12m)

Table 5.4: Performance of VISEN training algorithms with varying number of feature vectors per image (ω). Main time in seconds, approximate time also shown.

Figure 5.8 and Table 5.4 show, respectively, the mAP and codebook generation latency for this test. Precision peaks at $\omega = 1800$ for the *Traditional K-means* with 49%; the *Online* method peaks at $\omega = 1200$ with 43%. The difference in precision between both approaches is slim, while codebook generation latency is much lower for *Online K-means*, making it an attractive approach for the scheme's *operating phase*. With larger values for ω precision decreases – having too much chosen features per image eventually leads to noise, affecting results precision. Therefore, the ideal value for ω lies between 1200 and 1800.

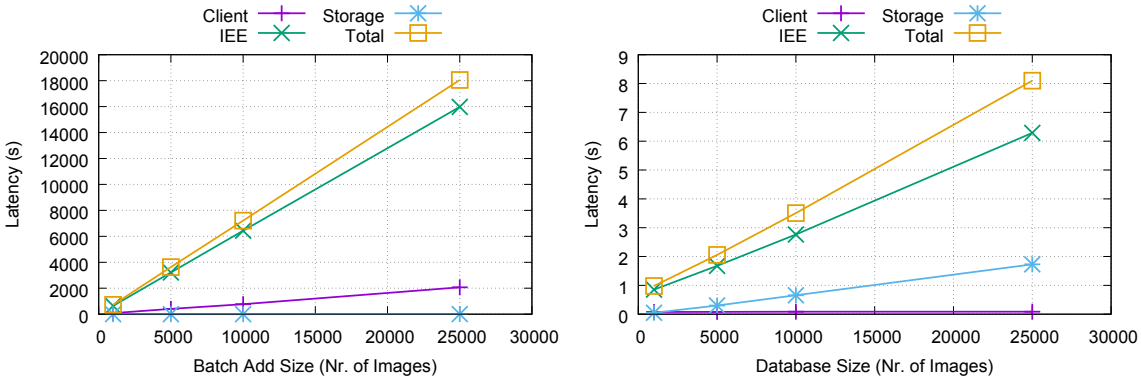
5.3.2 Evaluating the *Operating Phase*

In this batch of tests we evaluate the performance of the two main *operating phase* protocols: *Add* and *Search*. We performed these tests using a larger dataset than INRIA Holidays: the MIR Flickr dataset, which contains a total of 25 000 images.

In Section 5.3.2.1 we analyse the latency of both *Add* and *Search* protocols, and the impact of each protocol participant individually; in Sections 5.3.2.2 and 5.3.2.3 we measure the performance impact of varying the number of clusters and ω; to conclude, we measure the impact of varying ω on the number of *Storage* entries in Section 5.3.2.4.

Conversely to BISEN, we will present our latency plots in function of the number of images on the database, instead of the number of entries in the *Storage Service*; while BISEN's main data element was the keyword, we consider VISEN's database to be composed of atomic images (although images amount to keywords from the *Storage Service* perspective, the scheme only contemplates insertion and removal of full images). For reference, the full dataset of 25 000 created 4.6 million entries in *Storage* for $\omega = 1200$.

5.3.2.1 Performance of Individual Participants



a VISEN *Add* protocol.

b VISEN *Search* protocol.

Figure 5.9: Performance comparison of each participant for the *Add* and *Search* protocols of VISEN.

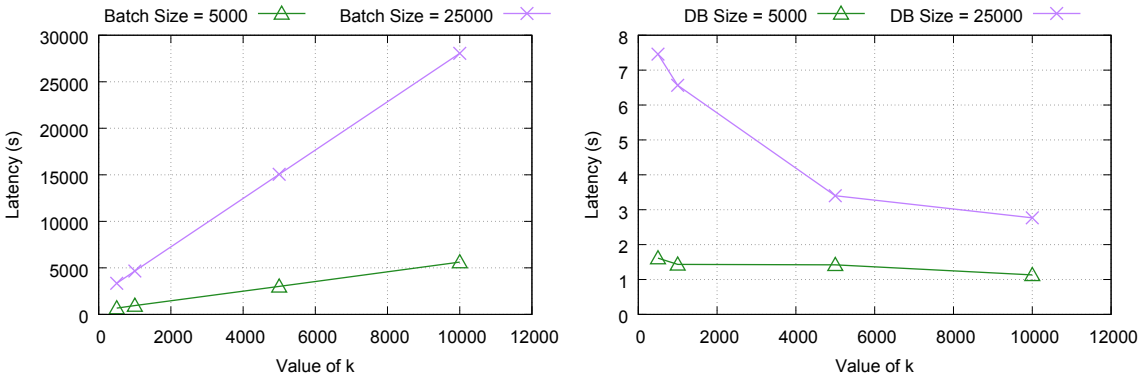
In this test we analyse the impact of each participant in our protocols, for a varying database size (from 500 to 25 000 images) and fixed $k = 5000$ and $\omega = 500$.

In Figure 5.9 we present the performance of each participant in the *Add* and *Search* protocols. In the *Add* protocol (Figure 5.9a) we observed a linear growth in latency with larger batches of images, a behaviour we expected (processing a single image is constant). The largest slice of processing is in the *IEE*: it is responsible for approximating feature vectors to clusters, and also to generate entries for the *Storage Service*, and respective messages. Processing on the *Client* (feature extraction) is essentially constant for each image – and thus, for increasingly more images it exhibits linear growth; *Storage Service* processing is minimal. On average, a single image takes 0.7 seconds to be added on current parameterisations.

In the *Search* protocol (Figure 5.9b), which takes a single image as input, the *Client*'s feature extraction step is constant, as with *Add*. Since searches are done against increasingly larger databases, a linear growth in latency in the *IEE* and *Storage Service* is expected. In particular, the *IEE* spends 10% of its processing approximating vectors to clusters, 68% of time generating labels and preparing requests to *Storage*, and 21% of time decoding and processing the respective responses; the rest of the time being spent in result scoring and generic processing. These results are similar to BISEN, since the *IEE* main bottleneck is also *Storage Service* interaction; in VISEN, it amounts for 89% of *IEE* processing time, while in BISEN it amounts to 96%.

5.3.2.2 Performance Impact of Varying the Number of Clusters

Figure 5.10 presents our test with fixed $\omega = 500$ and varying k (between 500 and 10 000), with two samples, for databases of 5 000 and 25 000 images. While *Add* shows a linear behaviour, due to the increasing number of comparisons that have to be made for each



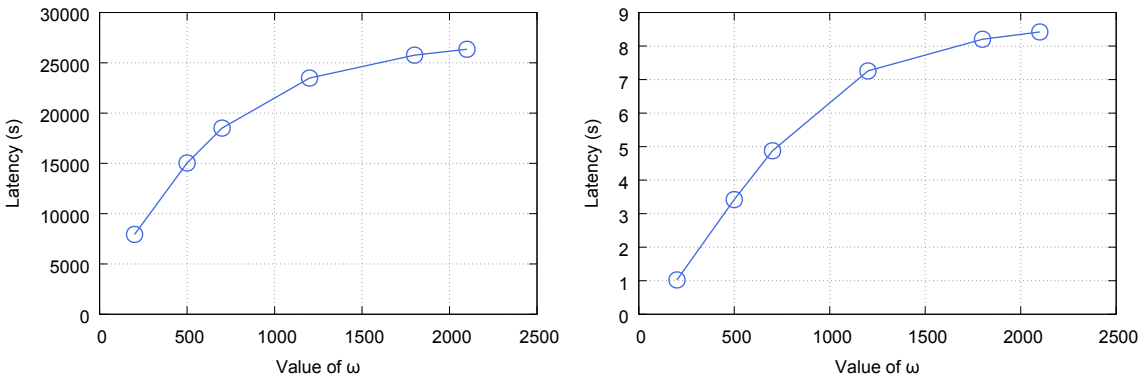
a *Add* protocol. b *Search* protocol.

Figure 5.10: Performance of VISEN under different values for k in the *Add* and *Search* protocols.

image, and increasing number of entries that have to be put in *Storage*, *Search* latency actually decreases with higher values of k.

While the number of comparisons still grows linearly with k (taking about 10% of *IEE* processing time), our intuition is as follows: a larger k implies feature vectors will be more disperse across clusters, *i.e.* each cluster will have less vectors (and, as such, less images) associated with it; therefore, and since the bottleneck in *IEE* processing is due to *Storage Service* interaction, if less images are associated with a given needed cluster, then less entries have to be retrieved from *Storage*. The advantage is twofold: results are more meaningful, since features are more compartmentalised, and *Search* latency is improved.

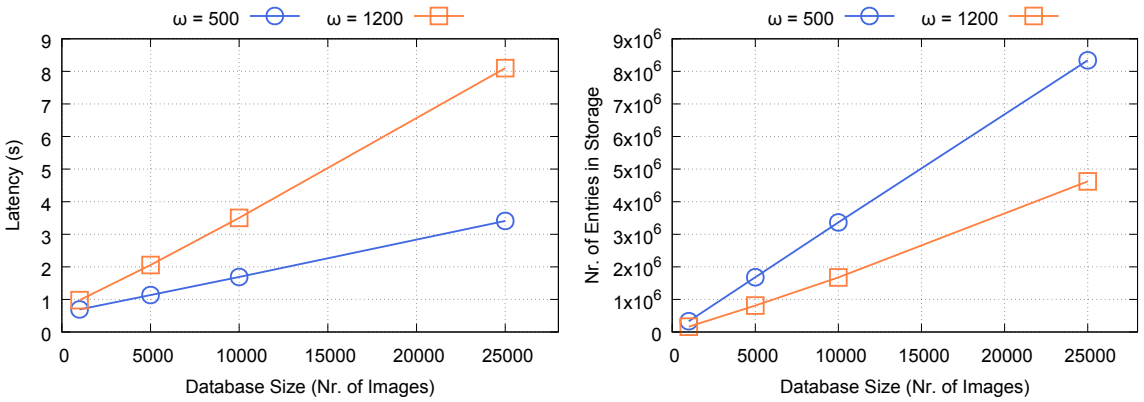
5.3.2.3 Performance Impact of Varying ω



a *Add* protocol. b *Search* protocol.

Figure 5.11: Performance of VISEN under different values for ω in the *Add* and *Search* protocols.

Figure 5.11 presents the latency of VISEN's *Add* and *Search* protocols under different values of ω. We fixed $k = 5000$ and the database size at the maximum value (25 000



a *Search* performance for $\omega \in [500, 1200]$.

b Number of entries in *Storage* for varying ω.

Figure 5.12: Effect of varying ω on VISEN *Search* performance and *Storage*.

images). We observed the cost of increasing the number of vectors is amortised for larger values. Nonetheless, although using more feature vectors per image only impacts performance sub-linearly, taking into account best precision is obtained with $\omega \approx 1200$, usage of larger values for ω is not a particularly interesting approach.

5.3.2.4 Comparing the Number of Storage Entries for Different ω Values

Figure 5.12 presents a comparison of performance and *Storage* overhead while varying $\omega \in [500, 1200]$ (fixed $k = 5000$ and varying database size). Having a higher ω increases *Search* latency (Figure 5.12a), while also producing less entries in the *Storage Service* (Figure 5.12b).

Our first intuition was that, for larger ω, the number of entries would be larger, as more feature vectors (and, therefore, visual words) would be produced – the number of visual words per image being limited by $min(k, \omega)$.

Nevertheless, larger values of ω produce the opposite effect, resulting in a database with less entries. Our first observation is that higher ω values produce images with less clusters associated to it, leading to higher cluster frequencies (*i.e.*, more vectors are clustered to the same centroids), but less entries overall. Our conclusion is that increasing ω results in the generation of similar feature vectors (*i.e.*, feature vectors describing the same image feature); hence, the insertion of less entries (with a higher average frequency per cluster) in the *Storage Service*.

Albeit the database being smaller with larger ω values, performance of the *Search* protocol still degrades. This result is correlated with the sparsity of feature vectors across clusters and the results from Section 5.3.2.2: since a higher ω implies more vectors are inserted into the same clusters, and that these clusters will tend to be more popular overall, retrieval of a single popular cluster will imply the retrieval of more entries (all the entries and respective images referring to that cluster).

Database Size (Nr. of images)	Add		Search	
	VISEN	**MuSE**	**VISEN**	**MuSE**
1 000	721	95	0.976	0.563
5 000	3 626	480	2.057	2.858
10 000	7 209	957	3.504	5.964
25 000	18 051	2405	8.099	14.942

Table 5.5: Latency comparison between VISEN and MuSE (Ferreira et al., 2018) schemes. All times are in seconds. Fixed $k = 5000$ and $\omega = 1200$.

5.3.3 Comparison with the State-of-the-Art

In this test we compare our solution to MuSE (Ferreira et al., 2018), a scheme for multi-modal searchable encryption – albeit here used only with its image component. Table 5.5 shows the comparison in latency between our scheme and MuSE (Ferreira et al., 2018), for the *Add* and *Search* protocols with fixed parameters $k = 5000$ and $\omega = 500$. On the one hand, MuSE performs better on its *Add* protocol, which can be explained by its simpler system model, and its lack of need for external memory storage – a large portion of VISEN processing is spent communicating between *IEE* and *Storage Service*. On the other hand, VISEN performs and scales better on searches, in particular for larger database sizes, which can be explained by the simpler, and fewer, indexing structures and lower processing time associated with them.

5.3.4 Discussion

VISEN evaluation allowed us to assess not only the scheme's performance, but also its precision. We were able to obtain satisfactory results for both, by implementing a scheme that scales linearly to its database size.

Our precision results, peaking at 49%, are similar to the ones found in the literature for BoVW retrieval models (Ferreira et al., 2018; Xia et al., 2016) based solely on feature clustering. Although orthogonal to the scope of this work, some techniques to improve precision of clustering schemes have been proposed (Jegou et al., 2008; Liu et al., 2014b), which include dimensionality-reduction procedures, such as Principal Component Analysis (Alpaydın, 2010, Section 6.3) or Self-Organising Maps (Alpaydın, 2010, Section 12.2.3). Low precision when training highly-dimensional data is a problem known in machine learning as the *curse of dimensionality* (Alpaydın, 2010, Section 8.3). Such techniques for dimensionality-reduction could be done by the *Client* together with the feature extraction step.

Our results for the two different k-means techniques show that, when compared to the traditional version, online k-means performs similarly precision-wise, while being both faster and having low memory requirements, which makes it ideal for *IEE* usage, and also for thin clients; in any case, CBIR systems usually consider training to happen only once – during system bootstrapping – which amortises the impact of such operation.

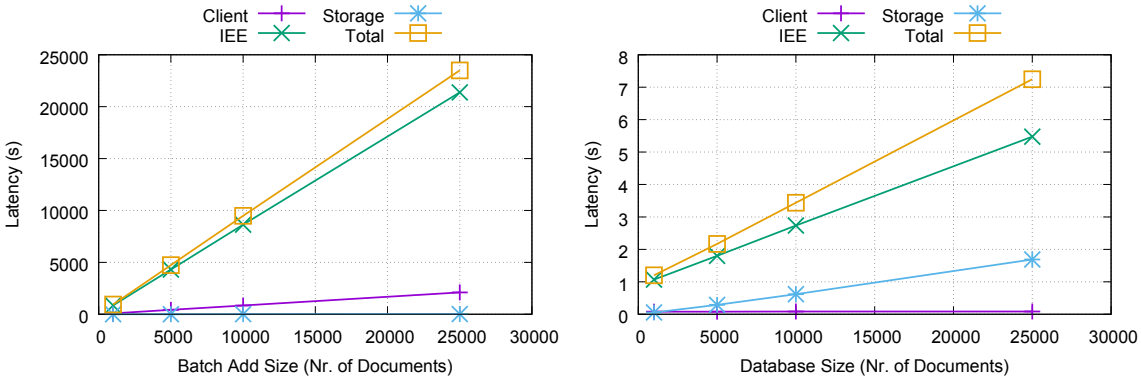
The third technique for clustering, LSH, showed worst precision; yet, it may have its advantages. Since it does not require any training, it might be interesting to use in shorter lived systems, where fast bootstrapping may be needed. Furthermore, in additional tests we performed, we obtained 20% precision for LSH with $k = 10000$ and $\omega = 2000$, which indicates further work might find the method to be usable.

Our results on the impact of varying the number of clusters, and the number of descriptors per image, also originated interesting results, showing that precision and performance have to be considered together when parameterising the scheme, resulting in a practical trade-off that only presented itself during the evaluation phase.

As future work, we would like to analyse precision by dynamically changing k and ω between *training* and *operating phases*. In current tests parameterisations are fixed throughout; performing training with better parameters (such as an ideal ω), and then using lower, more efficient, values for the *operating phase* might be an interesting evaluation direction to present a more practical scheme.

5.4 MISEN Evaluation

Since our implementation of MISEN does not consider any particular improvement, apart from integrating both BISEN and VISEN together, we simply present the performance of the *Add* and *Search* protocols in MISEN (Figure 5.13). For this evaluation we used the MIR Flickr dataset (Huiskes and Lew, 2008) which, together with the 25 000 images we used for VISEN evaluation, also contains textual data associated to each image. Queries were generated by randomly selecting images from the dataset, and combining random words from the respective text documents.



a MISEN *Add* protocol.

b MISEN *Search* protocol.

Figure 5.13: Performance comparison of each participant for the *Add* and *Search* protocols of MISEN.

Due to the nature of the dataset – small text documents (averaging ten keywords per document) – BISEN processing has low impact on the *Add* protocol, and therefore MISEN exhibits a similar behaviour to that of VISEN. In *Search* both BISEN and VISEN present

similar behaviour, with processing being spent mostly in the *IEE* – therefore, MISEN reflects that same behaviour.

Chapter 6

Conclusion

The problem of *computing over encrypted data* has become relevant in the context of Cloud computing and the current trend to outsource data to third-party servers. Adoption of these services by users, and the large amount of data produced nowadays, brings the need for efficient solutions to query and use outsourced data without performance penalties, while retaining strong security guarantees.

6.1 Conclusion

We have studied different approaches and the state-of-the-art on SSE, combined with an analysis on recent trusted hardware solutions. Current solutions for secure retrieval have made a trade-off between security, usability, and performance, usually sacrificing one of these dimensions in favour of others. By designing and implementing novel schemes for SSE we provide an answer to the question posed in this thesis: *how can we improve SSE solutions to provide better security guarantees, while also being practical and efficient?*

Our schemes for isolated searchable encryption provide an efficient and usable approach to the problem of *computing over encrypted data* by building a hybrid between classical SSE and modern trusted hardware, an approach still novel in the literature. We have designed schemes for Boolean retrieval over text data, Content-Based Image Retrieval (CBIR) over image data, and for multimodal data. The analysis and evaluation of these schemes shows them to improve security guarantees, while also providing better usability and linear scalability, thus providing satisfactory performance.

Considering a trust anchor on the Cloud server allows us to execute heavy computations remotely, reducing client and network overhead, and thus operation latency. The use of a single inverted index for data storage is central to our approach, as it provides constant access times without the loss of security. Moreover, given the IEE security model,

we can improve the state-of-the-art by providing better privacy and authentication guarantees, and reducing leakage. Our protocols achieve minimal leakage, in the sense that only accessed patterns in untrusted storage are revealed – but no information about their contents or relations; such leakage can only be improved through techniques such as Oblivious RAM (ORAM). Without it, inference patterns may form over time, and attackers can rely on external knowledge for help on discerning information about added and accessed entries on *Storage*. Nonetheless, even ORAM-based solutions are sensitive to inference attacks, due to differences in response length across messages (Bost and Fouque, 2017); by having fixed-width responses in the ranked versions of our schemes, we are able to prevent such attacks.

Moreover, depending on trusted hardware manufacturers is still a requirement for schemes based on it – which can be a deterrent in practice, as such hardware implementations are usually close-sourced. As such, they lack the *crowd-auditing* advantages of open-source tools, and require trust in its vendor for the non-placement of backdoors. While our approach purposefully relies on IEEs for its definitions, the eventual advent of open-sourced trusted hardware would present itself as a preferable alternative implementation-wise. To this effect, in this thesis we also designed a Framework API for the IEE model, which abstracts implementation details and provides utilities for IEE programmers.

6.2 Future Work

As future work we intend to extend our schemes for multiple clients and servers, providing not only the same security guarantees, but also fault resilience and load distribution capabilities.

Extending the schemes for multiple clients may rely on existing access control and public key schemes for authentication; considering multiple servers might present itself as a more interesting approach, as IEE resource limitations involve a more careful study on load distribution across an *IEE cluster*. A model considering a cluster of IEE-enabled devices opens the possibility to build general purpose distributed systems and applications with better security and performance guarantees, in an environment as transparent as possible, both programming and deployment-wise. Furthermore, combining multiple IEEs in a Cloud-of-Clouds environment is also a possible solution for thwarting denial-of-service attacks, where IEE devices could automatically detect and recover from such failures.

Furthermore, a distributed approach on both *IEE* and *Storage* components might also be key to solve the current bottleneck in our implementation and evaluation, which is due to the network interaction between these components. We expect to leverage a tree-like structure for multiple *IEE* nodes, where a main *IEE* would instruct several *IEE* modules to interact with *Storage* – with each *IEE* processing their results internally before contacting the root *IEE*, which would be responsible for the processing of the final result.

We also plan to extend our Framework implementation to allow for different IEE instantiations, in particular ARM TrustZone, while retaining the same abstraction level of an IEE to the programmer – thus providing a generic Framework for IEE-based programming in heterogeneous Cloud environments.

Bibliography

Aaron, B, D. E. Tamir, N. D. Rishe and A Kandel (Mar. 2014). "Dynamic Incremental K-means Clustering". In: *2014 International Conference on Computational Science and Computational Intelligence*. Vol. 1, pp. 308–313. DOI: 10.1109/CSCI.2014.60.

Agrawal, R., J. Kiernan, R. Srikant and Y. Xu (2004). "Order Preserving Encryption for Numeric Data". In: *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*. SIGMOD '04. New York, NY, USA: ACM, pp. 563–574. ISBN: 1-58113-859-8. DOI: 10.1145/1007568.1007632. URL: http://doi.acm.org/10.1145/1007568.1007632.

Alpaydın, E. (2010). *Introduction to Machine Learning*. Ed. by T. Dietterich. 2nd. London, United Kingdom: The MIT Press. ISBN: 978-0-262-01243-0.

Anati, I., S. Gueron, S. Johnson and V. Scarlata (2013). "Innovative Technology for CPU Based Attestation and Sealing". In: *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. New York, NY, USA: ACM, pp. 1–7. ISBN: 978-1-4503-2118-1. DOI: 10.1.1.405.8266.

Apple Insider (Feb. 2016). *Apple Music Passes 11M Subscribers as iCloud Hits 782M Users*. URL: http://appleinsider.com/articles/16/02/12/apple-music-passes-11m-subscribers-as-icloud-hits-782m-users.

Arasu, A., S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy and R. Venkatesan (Jan. 2013). "Orthogonal Security With Cipherbase". In: *6th Biennial Conference on Innovative Data Systems Research (CIDR'13)*. URL: https://www.microsoft.com/en-us/research/publication/orthogonal-security-with-cipherbase/.

ARM (2009). "ARM Security Technology: Building a Secure System using TrustZone Technology ARM". In: *ARM White Paper*. URL: https://tinyurl.com/armtrustzonereport.

Armbrust, M., A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica and M. Zaharia (Apr. 2010). "A View of Cloud Computing". In: *Communications of the ACM* 53.4, pp. 50–58. ISSN: 0001-0782. DOI: 10.1145/1721654.1721672. URL: http://doi.acm.org/10.1145/1721654.1721672.

Bagirov, A. M., J. Ugon and D. Webb (Apr. 2011). "Fast Modified Global K-means Algorithm for Incremental Cluster Construction". In: *Pattern Recognition* 44.4, pp. 866–876. ISSN: 0031-3203. DOI: 10.1016/j.patcog.2010.10.018. URL: http://dx.doi.org/10.1016/j.patcog.2010.10.018.

Bajaj, S. and R. Sion (2011). "TrustedDB: A Trusted Hardware Based Database with Privacy and Data Confidentiality". In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*. SIGMOD '11. New York, NY, USA: ACM, pp. 205–216. ISBN: 978-1-4503-0661-4. DOI: 10.1145/1989323.1989346. URL: http://doi.acm.org/10.1145/1989323.1989346.

Barbosa, M., B. Portela, G. Scerri and B. Warinschi (Mar. 2016). "Foundations of Hardware-Based Attested Computation and Application to SGX". In: *Proceedings of the 2016 IEEE European Symposium on Security and Privacy (Euro S&P)*, pp. 245–260. DOI: 10.1109/EuroSP.2016.28.

Bassil, Y. (2012). "A Survey on Information Retrieval, Text Categorization, and Web Crawling". In: *Journal of Computer Science and Research* 1.6, pp. 1–11. arXiv: 1212.2065. URL: http://arxiv.org/abs/1212.2065.

Baumann, A., M. Peinado and G. Hunt (2014). "Shielding Applications from an Untrusted Cloud with Haven". In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, pp. 267–283. ISBN: 978-1-931971-16-4. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/baumann.

Bay, H., A. Ess, T. Tuytelaars and L. Van Gool (June 2008). "Speeded-Up Robust Features (SURF)". In: *Computer Vision and Image Understanding* 110.3, pp. 346–359. ISSN: 1077-3142. DOI: 10.1016/j.cviu.2007.09.014. URL: http://dx.doi.org/10.1016/j.cviu.2007.09.014.

Bellare, M., A. Boldyreva and A. O'Neill (2007). "Deterministic and Efficiently Searchable Encryption". In: *Proceedings of the 27th Annual International Cryptology Conference on Advances in Cryptology*. CRYPTO'07. Berlin, Heidelberg: Springer-Verlag, pp. 535–552. ISBN: 3-540-74142-9, 978-3-540-74142-8. URL: http://dl.acm.org/citation.cfm?id=1777777.1777820.

Bellare, M., M. Fischlin, A. O'Neill and T. Ristenpart (2008). "Deterministic Encryption: Definitional Equivalences and Constructions Without Random Oracles". In: *Proceedings of the 28th Annual Conference on Cryptology: Advances in Cryptology*. CRYPTO 2008. Berlin, Heidelberg: Springer-Verlag, pp. 360–378. ISBN: 978-3-540-85173-8. DOI: 10.1007/978-3-540-85174-5_20. URL: http://dx.doi.org/10.1007/978-3-540-85174-5_20.

Bessani, A., M. Correia, B. Quaresma, F. André and P. Sousa (Nov. 2013). "DepSky: Dependable and Secure Storage in a Cloud-of-Clouds". In: *ACM Transactions on Storage* 9.4, 12:1–12:33. ISSN: 1553-3077. DOI: 10.1145/2535929. URL: http://doi.acm.org/10.1145/2535929.

Best, R. M. (1981). *Crypto Microprocessor for Executing Enciphered Programs*. URL: https://www.google.com/patents/US4278837.

Boldyreva, A. and N. Chenette (2011). "Order-Preserving Encryption Revisited: Improved Security Analysis and Alternative Solutions". In: *Advances in Cryptology – CRYPTO 2011* LNCS, Springer, pp. 578–595.

Boldyreva, A., N. Chenette, Y. Lee and A. O'Neill (2009). "Order-Preserving Symmetric Encryption". In: *Proceedings of the 28th Annual International Conference on Advances in Cryptology: The Theory and Applications of Cryptographic Techniques*. EUROCRYPT '09. Berlin, Heidelberg: Springer-Verlag, pp. 224–241. ISBN: 978-3-642-01000-2. DOI: 10.1007/978-3-642-01001-9_13. URL: http://dx.doi.org/10.1007/978-3-642-01001-9_13.

Boneh, D., E.-J. Goh and K. Nissim (2005). "Evaluating 2-DNF Formulas on Ciphertexts". In: *Proceedings of the 2nd International Conference on Theory of Cryptography*. TCC'05. Berlin, Heidelberg: Springer-Verlag, pp. 325–341. ISBN: 3-540-24573-1, 978-3-540-24573-5. DOI: 10.1007/978-3-540-30576-7_18. URL: http://dx.doi.org/10.1007/978-3-540-30576-7_18.

Boneh, D., K. Lewi, M. Raykova, A. Sahai, M. Zhandry and J. Zimmerman (2015). "Semantically Secure Order-Revealing Encryption: Multi-Input Functional Encryption Without Obfuscation". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 9057, pp. 563–594. ISSN: 16113349. DOI: 10.1007/978-3-662-46803-6_19.

Bösch, C., P. Hartel, W. Jonker and A. Peter (Aug. 2014). "A Survey of Provably Secure Searchable Encryption". In: *ACM Computing Surveys* 47.2, 18:1–18:51. ISSN: 0360-0300. DOI: 10.1145/2636328. URL: http://doi.acm.org/10.1145/2636328.

Bost, R. (2016). "Sophos: Forward Secure Searchable Encryption". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. New York, NY, USA: ACM, pp. 1143–1154. ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978303. URL: http://doi.acm.org/10.1145/2976749.2978303.

Bost, R. and P.-A. Fouque (2017). *Thwarting Leakage Abuse Attacks against Searchable Encryption – A Formal Approach and Applications to Database Padding*. Cryptology ePrint Archive, Report 2017/1060.

Bost, R., B. Minaud and O. Ohrimenko (2017). "Forward and Backward Private Searchable Encryption from Constrained Cryptographic Primitives". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. New York, NY, USA: ACM, pp. 1465–1482. ISBN: 978-1-4503-4946-8. DOI: 10.1145/3133956.3133980. URL: http://doi.acm.org/10.1145/3133956.3133980.

Bratus, S., N. D'Cunha, E. Sparks and S. W. Smith (2008). "TOCTOU, Traps, and Trusted Computing". In: *Proceedings of the 1st International Conference on Trusted Computing and Trust in Information Technologies: Trusted Computing - Challenges and Applications*. Trust '08. Berlin, Heidelberg: Springer-Verlag, pp. 14–32. ISBN: 978-3-540-68978-2. DOI: 10.1007/978-3-540-68979-9_2. URL: http://dx.doi.org/10.1007/978-3-540-68979-9_2.

Cao, N., C. Wang, M. Li, K. Ren and W. Lou (Jan. 2014). "Privacy-Preserving Multi-Keyword Ranked Search over Encrypted Cloud Data". In: *IEEE Transanctions on Parallel Distributed Systems* 25.1, pp. 222–233. ISSN: 1045-9219. DOI: 10.1109/TPDS.2013.45. URL: http://dx.doi.org/10.1109/TPDS.2013.45.

Cash, D., S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Roşu and M. Steiner (2013). "Highly-Scalable Searchable Symmetric Encryption with Support for Boolean Queries". In: *Advances in Cryptology – CRYPTO 2013*. Ed. by R. Canetti and J. A. Garay. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 353–373. ISBN: 978-3-642-40041-4.

Cash, D., J. Jaeger, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Roşu and M. Steiner (2014). "Dynamic Searchable Encryption in Very-Large Databases: Data Structures and Implementation". In: *Proceedings of the 2014 Network and Distributed System Security Symposium (NDSS '14)* February, pp. 1–32. DOI: 10.14722/ndss.2014.23264. URL: http://www.internetsociety.org/doc/dynamic-searchable-encryption-very-large-databases-data-structures-and-implementation.

Cash, D., P. Grubbs, J. Perry and T. Ristenpart (2015). "Leakage-Abuse Attacks Against Searchable Encryption". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. New York, NY, USA: ACM, pp. 668–679. ISBN: 978-1-4503-3832-5. DOI: 10.1145/2810103.2813700. URL: http://doi.acm.org/10.1145/2810103.2813700.

Checkoway, S. and H. Shacham (2013). "Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface". In: *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '13. New York, NY, USA: ACM, pp. 253–264. ISBN: 978-1-4503-1870-9. DOI: 10.1145/2451116.2451145. URL: http://doi.acm.org/10.1145/2451116.2451145.

Chenette, N., K. Lewi, S. A. Weis and D. J. Wu (2016). "Practical Order-Revealing Encryption with Limited Leakage". In: *Fast Software Encryption (FSE)*.

Chow, R., P. Golle, M. Jakobsson, E. Shi, J. Staddon, R. Masuoka and J. Molina (2009). "Controlling Data in the Cloud: Outsourcing Computation Without Outsourcing Control". In: *Proceedings of the 2009 ACM Workshop on Cloud Computing Security*. CCSW '09. New York, NY, USA: ACM, pp. 85–90. ISBN: 978-1-60558-784-4. DOI: 10.1145/1655008.1655020. URL: http://doi.acm.org/10.1145/1655008.1655020.

Columbus, L. (Apr. 2017a). *2017 State Of Cloud Adoption And Security*. URL: https://tinyurl.com/cloudforbes.

— (Apr. 2017b). *Roundup Of Cloud Computing Forecasts, 2017*. URL: https://tinyurl.com/forbescolumbus.

Cook, T. (2016). *A Message to Our Customers*. URL: https://www.apple.com/customer-letter/.

Costan, V. and S. Devadas (2016). "Intel SGX Explained". In: *Cryptology ePrint Archive, Report 2016/086*, p. 108.

Curtmola, R., J. Garay, S. Kamara and R. Ostrovsky (2006). "Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions". In: *Proceedings of the 13th ACM Conference on Computer and Communications Security*. CCS '06. New York, NY, USA: ACM, pp. 79–88. ISBN: 1-59593-518-5. DOI: 10.1145/1180405.1180417. URL: http://doi.acm.org/10.1145/1180405.1180417.

Darrow, B. (Mar. 2016). *Dropbox Claims Half a Billion Users*. URL: http://fortune.com/2016/03/07/dropbox-half-a-billion-users/.

Dijkstra, E. W. (1961). *ALGOL 60 Translation: An ALGOL 60 Translator for the X1 and Making a Translator for ALGOL 60*. Tech. rep. Amsterdam, The Netherlands: Stichting Mathematisch Centrum.

Dyer, J. G., M. Lindemann, R. Perez, R. Sailer, L. Van Doorn, S. W. Smith and S. Weingart (2001). "Building the IBM 4758 Secure Coprocessor". In: *Computer* 34.10, pp. 57–66. ISSN: 00189162. DOI: 10.1109/2.955100.

ElGamal, T. (July 1985). "A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms". In: *IEEE Transactions on Information Theory* 31.4, pp. 469–472. ISSN: 0018-9448. DOI: 10.1109/TIT.1985.1057074.

Ferreira, B. and H. Domingos (2013). "Searching Private Data in a Cloud Encrypted Domain". In: *Proceedings of the 10th Conference on Open Research Areas in Information Retrieval*. OAIR '13. Paris, France, France: Le Centre de Hautes Études Internationales d'Informatique Documentaire, pp. 165–172. ISBN: 978-2-905450-09-8. URL: http://dl.acm.org/citation.cfm?id=2491748.2491783.

Ferreira, B., J. Rodrigues, J. Leitão and H. Domingos (Sept. 2015). "Privacy-Preserving Content-Based Image Retrieval in the Cloud". In: *Proceedings of the 34th IEEE International Symposium on Reliable Distributed Systems (SRDS '15)*. Washington, DC, USA: IEEE, pp. 11–20. ISBN: 978-1-4673-9302-7. DOI: 10.1109/SRDS.2015.27. URL: http://dx.doi.org/10.1109/SRDS.2015.27.

Ferreira, B., J. Leitão and H. Domingos (June 2017). "Multimodal Indexable Encryption for Mobile Cloud-Based Applications". In: *Proceedings of the 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'17)*. IEEE, pp. 213–224. ISBN: 978-1-5386-0542-4. DOI: 10.1109/DSN.2017.31. URL: http://ieeexplore.ieee.org/document/8023124/.

— (2018). "MuSE: Multimodal Searchable Encryption for Cloud Applications". In: *Proceedings of the 37th IEEE International Symposium on Reliable Distributed Systems (SRDS'18)*. Washington, DC, USA: IEEE.

Fuhry, B., R. Bahmani, F. Brasser, F. Hahn, F. Kerschbaum and A.-R. Sadeghi (2017). "HardIDX: Practical and Secure Index with SGX". In: *Data and Applications Security and Privacy XXXI. DBSec 2017*. Vol. 10359 LNCS, pp. 386–408. ISBN: 9783319611754. DOI: 10.1007/978-3-319-61176-1_22. arXiv: 1703.04583. URL: http://link.springer.com/10.1007/978-3-319-61176-1_22.

Fung, B. (May 2015). *In 5 Years, 80 Percent of the Whole Internet Will Be Online Video*. URL: https://www.washingtonpost.com/news/the-switch/wp/2015/05/27/in-5-years-80-percent-of-the-whole-internet-will-be-online-video/.

Gentry, C. (2009). "A Fully Homomorphic Encryption Scheme". PhD Thesis. Stanford University.

Gentry, C. and S. Halevi (2011). "Implementing Gentry's Fully-Homomorphic Encryption Scheme". In: *Proceedings of the 30th Annual International Conference on Theory and*

Applications of Cryptographic Techniques: Advances in Cryptology. EUROCRYPT'11. Berlin, Heidelberg: Springer-Verlag, pp. 129–148. ISBN: 978-3-642-20464-7. URL: http://dl.acm.org/citation.cfm?id=2008684.2008697.

Goh, E.-J., H. Shacham, N. Modadugu and D. Boneh (2003). "SiRiUS: Securing Remote Untrusted Storage". In: *Proceedings of the 10th Annual Network and Distributed System Security Symposium - NDSS '03* 0121481, pp. 131–145.

Goldreich, O. (1987). "Towards a Theory of Software Protection and Simulation by Oblivious RAMs". In: *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*. STOC '87. New York, NY, USA: ACM, pp. 182–194. ISBN: 0-89791-221-7. DOI: 10.1145/28395.28416. URL: http://doi.acm.org/10.1145/28395.28416.

Goldreich, O. and R. Ostrovsky (May 1996). "Software Protection and Simulation on Oblivious RAMs". In: *Journal of the ACM* 43.3, pp. 431–473. ISSN: 0004-5411. DOI: 10.1145/233551.233553. URL: http://doi.acm.org/10.1145/233551.233553.

Golle, P., J. Staddon and B. R. Waters (2004). "Secure Conjunctive Keyword Search over Encrypted Data". In: *Proceedings of the 2004 International Conference on Applied Cryptography and Network Security (ACNS '04)*. Yellow Mountains, China: Springer, pp. 31–45.

Google (2010). *BigQuery*. URL: https://cloud.google.com/bigquery/ (visited on 03/09/2018).

— (2016). *Encrypted BigQuery*. URL: https://github.com/google/encrypted-bigquery-client (visited on 03/08/2018).

Greenwald, G. and E. MacAskill (June 2013). *NSA Prism Program Taps in to User Data of Apple, Google and Others*. URL: https://www.theguardian.com/world/2013/jun/06/us-tech-giants-nsa-data.

HCA News (Sept. 2018). *Yes, Healthcare's Data Breach Problem Really Is That Bad*. URL: https://www.hcanews.com/news/yes-healthcares-data-breach-problem-really-is-that-bad.

Hoekstra, M., R. Lal, P. Pappachan, V. Phegade and J. Del Cuvillo (2013). "Using Innovative Instructions to Create Trustworthy Software Solutions". In: *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. HASP '13. New York, NY, USA: ACM, 11:1–11:1. ISBN: 978-1-4503-2118-1. DOI: 10.1145/2487726.2488370. URL: http://doi.acm.org/10.1145/2487726.2488370.

Hough, A. (2010). *Google Engineer Fired for Privacy Breach after 'Stalking and Harrassing Teenagers'*. URL: http://bit.ly/2o2VZh8.

Hsu, C, C Lu and S Pei (Nov. 2012). "Image Feature Extraction in Encrypted Domain With Privacy-Preserving SIFT". In: *IEEE Transactions on Image Processing* 21.11, pp. 4593–4607. ISSN: 1057-7149. DOI: 10.1109/TIP.2012.2204272.

Huiskes, M. J. and M. S. Lew (2008). "The MIR Flickr Retrieval Evaluation". In: *Proceedings of the 1st ACM International Conference on Multimedia Information Retrieval*. MIR '08. New York, NY, USA: ACM, pp. 39–43. ISBN: 978-1-60558-312-9. DOI: 10.1145/1460096.1460104. URL: http://doi.acm.org/10.1145/1460096.1460104.

Intel (2017). *Intel Software Guard Extensions SDK for Linux Developer Reference*.

— (2018a). *Intel SGX SDK for Linux Repository*. URL: https://github.com/01org/linux-sgx (visited on 14/06/2018).

— (2018b). *Performance Considerations for Intel Software Guard Extensions (Intel SGX) Applications*. Tech. rep. Intel.

Jegou, H., M. Douze and C. Schmid (2008). "Hamming Embedding and Weak Geometry Consistency for Large Scale Image Search". In: *Proceedings of the 10th European Conference on Computer Vision* October, pp. 304–317. ISSN: 03029743. DOI: 10.1007/978-3-540-88682-2_24.

Jeon, J., V. Lavrenko and R. Manmatha (2003). "Automatic Image Annotation and Retrieval Using Cross-Media Relevance Models". In: *Proceedings of the 26th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR '03. New York, NY, USA: ACM, pp. 119–126. ISBN: 1-58113-646-3. DOI: 10.1145/860435.860459. URL: http://doi.acm.org/10.1145/860435.860459.

Jing, Y., D. Liu, D. Kislyuk, A. Zhai, J. Xu, J. Donahue and S. Tavel (2015). "Visual Search at Pinterest". In: *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '15. New York, NY, USA: ACM, pp. 1889–1898. ISBN: 978-1-4503-3664-2. DOI: 10.1145/2783258.2788621. URL: http://doi.acm.org/10.1145/2783258.2788621.

Jones, K. S. (1972). "A Statistical Interpretation of Term Specificity and Its Application in Retrieval". In: *Journal of Documentation* 28, pp. 11–21.

Kamara, S. and T. Moataz (2017). *Boolean Searchable Symmetric Encryption with Worst-Case Sub-Linear Complexity*. Cryptology ePrint Archive, Report 2017/126. URL: https://eprint.iacr.org/2017/126.

Kamara, S., C. Papamanthou and T. Roeder (2012). "Dynamic Searchable Symmetric Encryption". In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. CCS '12. New York, NY, USA: ACM, pp. 965–976. ISBN: 978-1-4503-1651-4. DOI: 10.1145/2382196.2382298. URL: http://doi.acm.org/10.1145/2382196.2382298.

Kato, T. (Apr. 1992). *Database Architecture for Content-Based Image Retrieval*. Ed. by A. A. Jamberdino and C. W. Niblack. DOI: 10.1117/12.58497. URL: http://dx.doi.org/10.1117/12.58497.

Katz, J. and Y. Lindell (2007). *Introduction to Modern Cryptography*. Ed. by D. Stinson. 1st. Boca Raton, Florida: Chapman & Hall/CRC. ISBN: 1584885513.

Kauer, B. (2007). "OSLO: Improving the Security of Trusted Computing". In: *Proceedings of 16th USENIX Security Symposium*. SS'07. Berkeley, CA, USA: USENIX Association, 16:1–16:9. ISBN: 111 333 5555 77 9. URL: http://dl.acm.org/citation.cfm?id=1362903.1362919.

Kerschbaum, F. (2015). "Frequency-Hiding Order-Preserving Encryption". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*.

CCS '15. New York, NY, USA: ACM, pp. 656–667. ISBN: 978-1-4503-3832-5. DOI: 10.1145/2810103.2813629. URL: http://doi.acm.org/10.1145/2810103.2813629.

Khalaf, S. (June 2014). *Health and Fitness Apps Finally Take Off, Fueled by Fitness Fanatics*. URL: http://flurrymobile.tumblr.com/post/115192181465/health-and-fitness-apps-finally-take-off-fueled.

Kocher, P., D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz and Y. Yarom (Jan. 2018). "Spectre Attacks: Exploiting Speculative Execution". In: *ArXiv e-prints*. arXiv: 1801.01203.

Kursawe, K., D. Schellekens and B. Preneel (2005). "Analyzing Trusted Platform Communication". In: *ECRYPT Workshop, CRASH – CRyptographic Advances in Secure Hardware*, p. 8.

Lardinois, F. (Feb. 2016). *Gmail Now Has More Than 1B Monthly Active Users*. URL: https://techcrunch.com/2016/02/01/gmail-now-has-more-than-1b-monthly-active-users/.

Lewis, D. (Sept. 2014). *iCloud Data Breach: Hacking And Celebrity Photos*. URL: https://tinyurl.com/forbesicloud.

Liu, C., L. Zhu, M. Wang and Y.-A. Tan (May 2014a). "Search Pattern Leakage in Searchable Encryption: Attacks and New Construction". In: *Information Sciences - Informatics and Computer Science, Intelligent Systems, Applications* 265, pp. 176–188. ISSN: 0020-0255. DOI: 10.1016/j.ins.2013.11.021. URL: https://doi.org/10.1016/j.ins.2013.11.021.

Liu, Z, H Li, L Zhang, W Zhou and Q Tian (May 2014b). "Cross-Indexing of Binary SIFT Codes for Large-Scale Image Search". In: *IEEE Transactions on Image Processing* 23.5, pp. 2047–2057. ISSN: 1057-7149. DOI: 10.1109/TIP.2014.2312283.

Lloyd, S (Mar. 1982). "Least Squares Quantization in PCM". In: *IEEE Transactions on Information Theory* 28.2, pp. 129–137. ISSN: 0018-9448. DOI: 10.1109/TIT.1982.1056489.

Loi, T. L., J Heo, J Lee and S Yoon (Nov. 2013). "VLSH: Voronoi-based Locality Sensitive Hashing". In: *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 5345–5352. DOI: 10.1109/IROS.2013.6697130.

Lowe, D. G. (Nov. 2004). "Distinctive Image Features from Scale-Invariant Keypoints". In: *International Journal of Computer Vision* 60.2, pp. 91–110. ISSN: 0920-5691. DOI: 10.1023/B:VISI.0000029664.99615.94. URL: https://doi.org/10.1023/B:VISI.0000029664.99615.94.

Lu, W., A. Swaminathan, A. L. Varna and M. Wu (2009). "Enabling Search over Encrypted Multimedia Databases". In: *Information Technology Journal* 13, pp. 824–831. ISSN: 0277786X. DOI: 10.1117/12.806980. URL: http://proceedings.spiedigitallibrary.org/proceeding.aspx?doi=10.1117/12.806980.

MacQueen, J. B. (1967). "Some Methods for Classification and Analysis of MultiVariate Observations". In: *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics*

and Probability. Ed. by L. M. L. Cam and J Neyman. Vol. 1. University of California Press, pp. 281–297.

Manning, C. D., P. Raghavan and H. Schütze (2008). *Introduction to Information Retrieval*. New York, NY, USA: Cambridge University Press. ISBN: 0521865719, 9780521865715.

McKeen, F., I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue and U. R. Savagaonkar (2013). "Innovative Instructions and Software Model for Isolated Execution". In: *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. HASP '13. New York, NY, USA: ACM, 10:1–10:1. ISBN: 978-1-4503-2118-1. DOI: 10.1145/2487726.2488368. URL: http://doi.acm.org/10.1145/2487726.2488368.

Meeker, M. (2017). *Internet Trends 2017*. Tech. rep.

Microsoft (2018). *HealthVault*. URL: https://www.healthvault.com/ (visited on 08/05/2018).

Mikolajczyk, K. and T. Tuytelaars (2009). "Local Image Features". In: *Encyclopedia of Biometrics*. Ed. by S. Z. Li and A. Jain. Boston, MA: Springer US, pp. 939–943. ISBN: 978-0-387-73003-5. DOI: 10.1007/978-0-387-73003-5_224. URL: https://doi.org/10.1007/978-0-387-73003-5_224.

MongoDB (2018). *Encryption at Rest*. URL: https://docs.mongodb.com/manual/core/security-encryption-at-rest/ (visited on 28/08/2018).

Mourão, A., F. Martins and J. Magalhães (2013). "NovaSearch at TREC 2013 Federated Web Search Track: Experiments with Rank Fusion". In: *Proceedings of the 22nd Text REtrieval Conference (TREC 2013)*.

MySQL (2018). *MySQL Enterprise Transparent Data Encryption*. URL: https://www.mysql.com/products/enterprise/tde.html (visited on 29/08/2018).

Nistér, D. and H. Stewénius (2006). "Scalable Recognition with a Vocabulary Tree". In: *Proceedings of the 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Volume 2*. CVPR '06. Washington, DC, USA: IEEE Computer Society, pp. 2161–2168. ISBN: 0-7695-2597-0. DOI: 10.1109/CVPR.2006.264. URL: http://dx.doi.org/10.1109/CVPR.2006.264.

O'Hara, M. E. (May 2017). *Thousands of Patient Records Leaked in New York Hospital Data Breach*. URL: https://www.nbcnews.com/news/us-news/thousands-patient-records-leaked-hospital-data-breach-n756981.

Ohrimenko, O., F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani and M. Costa (2016). "Oblivious Multi-Party Machine Learning on Trusted Processors". In: *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, pp. 619–636. ISBN: 978-1-931971-32-4. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/ohrimenko.

O'Keeffe, D., D. Muthukumaran, P.-L. Aublin, F. Kelbert, C. Priebe, J. Lind, H. Zhu and P. Pietzuch (2018). *SGXSpectre*. URL: https://github.com/lsds/spectre-attack-sgx (visited on 23/03/2018).

Paillier, P. (1999). "Public-Key Cryptosystems Based on Composite Degree Residuosity Classes". In: *Proceedings of the 17th International Conference on Theory and Application of Cryptographic Techniques*. EUROCRYPT'99. Berlin, Heidelberg: Springer-Verlag, pp. 223–238. ISBN: 3-540-65889-0. URL: http://dl.acm.org/citation.cfm?id=1756123.1756146.

Pandey, O. and Y. Rouselakis (2012). "Property Preserving Symmetric Encryption". In: *Advances in Cryptology – EUROCRYPT 2012*. Ed. by D. Pointcheval and T. Johansson. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 375–391. ISBN: 978-3-642-29011-4.

Poddar, R., T. Boelter and R. A. Popa (2016). *Arx: A Strongly Encrypted Database System*. Tech. rep. URL: https://eprint.iacr.org/2016/591.

Popa, R. A., C. M. S. Redfield, N. Zeldovich and H. Balakrishnan (2011). "CryptDB: Protecting Confidentiality with Encrypted Query Processing". In: *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*. SOSP '11. New York, NY, USA: ACM, pp. 85–100. ISBN: 978-1-4503-0977-6. DOI: 10.1145/2043556.2043566. URL: http://doi.acm.org/10.1145/2043556.2043566.

Popa, R. A., F. H. Li and N. Zeldovich (2013). "An Ideal-Security Protocol for Order-Preserving Encoding". In: *Proceedings of the 2013 IEEE Symposium on Security and Privacy*. SP '13. Washington, DC, USA: IEEE Computer Society, pp. 463–477. ISBN: 978-0-7695-4977-4. DOI: 10.1109/SP.2013.38. URL: http://dx.doi.org/10.1109/SP.2013.38.

Popa, R. A., E. Stark, S. Valdez, J. Helfer, N. Zeldovich and H. Balakrishnan (2014). "Building Web Applications on Top of Encrypted Data Using Mylar". In: *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, pp. 157–172. ISBN: 978-1-931971-09-6. URL: https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/popa.

Priebe, C., K. Vaswani and M. Costa (May 2018). "EnclaveDB - A Secure Database using SGX". In: *Proceedings of the 39th IEEE Symposium on Security & Privacy (S&P '18)*. IEEE. URL: https://www.microsoft.com/en-us/research/publication/enclavedb-a-secure-database-using-sgx/.

Reinsel, D., J. Gantz and J. Rydning (2017). "Data Age 2025: The Evolution of Data to Life-Critical". In: *IDC White Paper* April, pp. 1–25. URL: https://www.seagate.com/files/www-content/our-story/trends/files/Seagate-WP-DataAge2025-March-2017.pdf.

RightScale (2018). *State of the Cloud Report*. Tech. rep.

Rivest, R., A. Shamir and L. Adleman (Feb. 1978a). "A Method for Obtaining Digital Signatures and Public-key Cryptosystems". In: *Communications of the ACM* 21.2, pp. 120–126. ISSN: 0001-0782. DOI: 10.1145/359340.359342. URL: http://doi.acm.org/10.1145/359340.359342.

Rivest, R. L., L. Adleman and M. L. Dertouzos (1978b). "On Data Banks and Privacy Homomorphisms". In: *Foundations of Secure Computation*, pp. 169–180. DOI: 10.1.1.

480.3392. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.500.3989&rep=rep1&type=pdf.

Roston, B. A. (Oct. 2017). *Database Leaks 47GB of Medical Records Including Names and Test Results*. URL: https://www.slashgear.com/database-leaks-47gb-of-medical-records-including-names-and-test-results-10503457/.

Rushe, D. (Aug. 2013). *Google: Don't Expect Privacy When Sending to Gmail*. New York, NY, USA. URL: https://www.theguardian.com/technology/2013/aug/14/google-gmail-users-privacy-email-lawsuit.

Salton, G, A Wong and C. S. Yang (Nov. 1975). "A Vector Space Model for Automatic Indexing". In: *Communications of the ACM* 18.11, pp. 613–620. ISSN: 0001-0782. DOI: 10.1145/361219.361220. URL: http://doi.acm.org/10.1145/361219.361220.

Santos, N., K. P. Gummadi and R. Rodrigues (2009). "Towards Trusted Cloud Computing". In: *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing*. HotCloud'09. Berkeley, CA, USA: USENIX Association. URL: http://dl.acm.org/citation.cfm?id=1855533.1855536.

Schuster, F., M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz and M. Russinovich (2015). "VC3: Trustworthy Data Analytics in the Cloud Using SGX". In: *Proceedings of the 2015 IEEE Symposium on Security and Privacy*. SP '15. Washington, DC, USA: IEEE Computer Society, pp. 38–54. ISBN: 978-1-4673-6949-7. DOI: 10.1109/SP.2015.10. URL: http://dx.doi.org/10.1109/SP.2015.10.

Seo, J., B. Lee, S. Kim and M.-W. Shih (2017). "SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs". In: *Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS '17)*. March.

Shih, M.-W., S. Lee, T. Kim and M. Peinado (2017). "T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs". In: *Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS '17)*. March.

Sion, R. (2009). "Trusted Hardware". In: *Encyclopedia of Database Systems*. Ed. by L. LIU and M. T. ÖZSU. Boston, MA: Springer US, pp. 3191–3192. ISBN: 978-0-387-39940-9. DOI: 10.1007/978-0-387-39940-9_1491. URL: https://doi.org/10.1007/978-0-387-39940-9_1491.

Song, D. X., D. Wagner and A. Perrig (2000). "Practical Techniques for Searches on Encrypted Data". In: *Proceedings of the 2000 IEEE Symposium on Security and Privacy*. SP '00. Washington, DC, USA: IEEE Computer Society, pp. 44–55. ISBN: 0-7695-0665-8. DOI: 10.1109/SECPRI.2000.848445. URL: http://dl.acm.org/citation.cfm?id=882494.884426.

Sparks, E. (2007). *TPM Reset Attack*. URL: https://web.archive.org/web/20171224141030/http://www.cs.dartmouth.edu/~pkilab/sparks/ (visited on 22/09/2018).

Stallings, W. and L. Brown (2014). *Computer Security: Principles and Practice*. 3rd. Upper Saddle River, NJ, USA: Prentice Hall Press. ISBN: 9781292066172.

Stefanov, E., E. Shi and D. Song (2012). "Towards Practical Oblivious RAM". In: *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS '12)*. arXiv: 1106.3652. URL: http://arxiv.org/abs/1106.3652.

Stefanov, E., M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu and S. Devadas (2013). "Path ORAM: An Extremely Simple Oblivious RAM Protocol". In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*. CCS '13. New York, NY, USA: ACM, pp. 299–310. ISBN: 978-1-4503-2477-9. DOI: 10.1145/2508859.2516660. URL: http://doi.acm.org/10.1145/2508859.2516660.

Stefanov, E., C. Papamanthou and E. Shi (2014). "Practical Dynamic Searchable Encryption with Small Leakage". In: *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS '14)*. ISBN: 1891562355. URL: http://dl.acm.org/citation.cfm?id=2636328.

Swain, M. J. and D. H. Ballard (Nov. 1991). "Color Indexing". In: *International Journal of Computer Vision* 7.1, pp. 11–32. ISSN: 0920-5691. DOI: 10.1007/BF00130487. URL: http://dx.doi.org/10.1007/BF00130487.

Swaminathan, A., Y. Mao, G.-M. Su, H. Gou, A. L. Varna, S. He, M. Wu and D. W. Oard (2007). "Confidentiality-Preserving Rank-Ordered Search". In: *Proceedings of the 2007 ACM Workshop on Storage Security and Survivability*. StorageSS '07. New York, NY, USA: ACM, pp. 7–12. ISBN: 978-1-59593-891-6. DOI: 10.1145/1314313.1314316. URL: http://doi.acm.org/10.1145/1314313.1314316.

Teufel, S. (2007). "An Overview of Evaluation Methods in TREC Ad Hoc Information Retrieval and TREC Question Answering". In: *Evaluation of Text and Speech Systems*, pp. 163–186.

Thomas, J. (2017). "Searchability". In: *Nineteenth-Century Illustration and the Digital*. Cham: Springer International Publishing, pp. 33–64. ISBN: 978-3-319-58148-4. DOI: 10.1007/978-3-319-58148-4_3. URL: http://link.springer.com/10.1007/978-3-319-58148-4_3.

Titcomb, J. (Nov. 2016). *Mobile Web Usage Overtakes Desktop for First Time*. URL: http://www.telegraph.co.uk/technology/2016/11/01/mobile-web-usage-overtakes-desktop-for-first-time/.

Trusted Computing Group (2009). *ISO/IEC 11889-1:2009 — Trusted Platform Module*. Tech. rep. International Organisation for Standardisation.

— (2015). *ISO/IEC 11889-1:2015(E) — Trusted Platform Module Library*. Tech. rep. International Organisation for Standardisation.

Turner, K. (Sept. 2016). *Hacked Dropbox Login Data of 68 Million Users is Now for Sale on the Dark Web*. URL: https://tinyurl.com/wpostdropbox.

Wang, C., N. Cao, K. Ren and W. Lou (Aug. 2012). "Enabling Secure and Efficient Ranked Keyword Search over Outsourced Cloud Data". In: *IEEE Transactions on Parallel Distributed Systems* 23.8, pp. 1467–1479. ISSN: 1045-9219. DOI: 10.1109/TPDS.2011.282. URL: http://dx.doi.org/10.1109/TPDS.2011.282.

Wang, J. Z., J. Li and G Wiederhold (Sept. 2001). "SIMPLIcity: Semantics-Sensitive Integrated Matching for Picture Libraries". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 23.9, pp. 947–963. ISSN: 0162-8828. DOI: 10.1109/34.955109.

Wang, X. S., K. Nayak, C. Liu, T.-H. H. Chan, E. Shi, E. Stefanov and Y. Huang (2014). "Oblivious Data Structures". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS '14. New York, NY, USA: ACM, pp. 215–226. ISBN: 978-1-4503-2957-6. DOI: 10.1145/2660267.2660314. URL: http://doi.acm.org/10.1145/2660267.2660314.

White, S. R. (Apr. 1987). "ABYSS: A Trusted Architecture for Software Protection". In: *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pp. 38–51. DOI: 10.1109/SP.1987.10021.

Wikipedia (2018). *Wikipedia Database Download*. URL: https://en.wikipedia.org/wiki/Wikipedia:Database_download (visited on 12/09/2018).

Wright, C. P., J. Dave and E. Zadok (2003). "Cryptographic File Systems Performance: What You Don't Know Can Hurt You". In: *Proceedings of the 2nd IEEE International Security in Storage Workshop*. SISW '03. Washington, DC, USA: IEEE Computer Society, pp. 47–54. ISBN: 0-7695-2059-6. URL: http://dl.acm.org/citation.cfm?id=998686.1007018.

Xia, Z., X. Wang, L. Zhang, Z. Qin, X. Sun and K. Ren (Nov. 2016). "A Privacy-Preserving and Copy-Deterrence Content-Based Image Retrieval Scheme in Cloud Computing". In: *IEEE Transactions on Information Forensics and Security* 11.11, pp. 2594–2608. ISSN: 1556-6013. DOI: 10.1109/TIFS.2016.2590944. URL: https://doi.org/10.1109/TIFS.2016.2590944.

Xu, Y., W. Cui and M. Peinado (2015). "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems". In: *Proceedings of the 2015 IEEE Symposium on Security and Privacy*. SP '15. Washington, DC, USA: IEEE Computer Society, pp. 640–656. ISBN: 978-1-4673-6949-7. DOI: 10.1109/SP.2015.45. URL: http://dx.doi.org/10.1109/SP.2015.45.

Yee, B. (1994). "Using Secure Coprocessors". PhD Thesis. Carnegie Mellon University.

Zhang, Y., J. Katz and C. Papamanthou (2016). "All Your Queries Are Belong to Us: The Power of File-Injection Attacks on Searchable Encryption". In: *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, pp. 707–720. ISBN: 978-1-931971-32-4. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/zhang.

Zobel, J. and A. Moffat (July 2006). "Inverted Files for Text Search Engines". In: *ACM Computing Surveys* 38.2. ISSN: 0360-0300. DOI: 10.1145/1132956.1132959. URL: http://doi.acm.org/10.1145/1132956.1132959.

Appendix: Framework Libraries API

```
1  namespace iee_util {
2      // thread pool
3      const unsigned thread_get_count();
4      int thread_add_work(void* (*task)(void*), void* args);
5      void thread_do_work();
6
7      // secure file i/o
8      void* open_secure(const char* name, const char* mode);
9      size_t write_secure(const void* ptr, size_t size, size_t count, void*
           stream);
10     size_t read_secure(void* ptr, size_t size, size_t count, void* stream);
11     void close_secure(void* stream);
12 }
```

Listing A.1: *IEE* library `iee_util`.

```
namespace iee_crypto {
    void random(void* out, size_t len);
    unsigned random_uint();
    unsigned random_uint_range(unsigned min, unsigned max);

    int sha256(void* out, const void* in, size_t len);
    int hmac_sha256(void* out, const void* in, const size_t in_len, const void
        * key, const size_t key_len);

    // symmetric authenticated cipher
    int encrypt_authenticated(void* out, const void* in, const size_t in_len,
        const void* nonce, const void* key);
    int decrypt_authenticated(void* out, const void* in, const size_t in_len,
        const void* nonce, const void* key);
}
```

Listing A.2: *IEE* library iee_crypto.

```
namespace outside_util {
    double time_elapsed_ms(struct timeval start, struct timeval end);

    int init_server(const int server_port);

    int socket_connect(const char* server_name, const int server_port);
    void socket_send(int socket, const void* buff, size_t len);
    void socket_receive(int socket, void* buff, size_t len);
}
```

Listing A.3: Outside library outside_util.

```
namespace sec_channel_client {
    void init_secure_connection(void* conn, const char* server_name, const int
         server_port);
    void socket_secure_send(void* conn, const void* buff, size_t len);
    void socket_secure_receive(void* conn, void* buff, size_t len);
    void close_secure_connection(void* conn);
}

namespace sec_channel_iee {
    void serve_secure_connection();
}
```

Listing A.4: Secure channel primitives sec_channel_client and sec_channel_iee.

Appendix B

Appendix: BISEN Evaluation Queries

0. time
1. person
2. year
3. way
4. day
5. thing
6. man
7. world
8. life
9. hand
10. part
11. child
12. history
13. country
14. born
15. lisbon
16. york

17. paris
18. time && person
19. time && person && year && way && day
20. time && person && year && way && day && thing && man && world && life && hand
21. time || person
22. time || person || year || way || day
23. time || person || year || way || day || thing || man || world || life || hand
24. (time && person) || (year && way)
25. (time && person) || (year && way) || (day && thing) || (man && world)
26. (time && person) || (year && way) || (day && thing) || (man && world) || (life && hand) || (part && child)
27. (time || person) && (year || way)
28. (time || person) && (year || way) && (day || thing) && (man || world)
29. (time || person) && (year || way) && (day || thing) && (man || world) && (life || hand) && (part || child)
30. !time && person && year && way && day && thing && man && world && life && hand
31. !time && !person && !year && !way && !day && thing && man && world && life && hand
32. !time && !person && !year && !way && !day && !thing && !man && !world && !life && !hand
33. !(time && person && year && way && day && thing && man && world && life && hand)
34. !time || !person || !year || !way || !day || !thing || !man || !world || !life || !hand